```
>>> A.upper_triangular(-1)
Matrix([
[1, 1, 1, 1],
[1, 1, 1, 1],
[0, 1, 1, 1],
[0, 0, 1, 1]])
```

values()

Return non-zero values of self.

vec()

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

Examples

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

See also:

vech (page 1357)

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

Reshapes the matrix into a column vector by stacking the elements in the lower triangle.

Parameters

diagonal : bool, optional

If `True`, it includes the diagonal elements.

check_symmetry : bool, optional

If `True`, it checks whether the matrix is symmetric.

Examples

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

Notes

This should work for symmetric matrices and vech can represent symmetric matrices in vector form with less size than vec.

See also:

vec (page 1357)

classmethod vstack(**args*)

Return a matrix formed by joining args vertically (i.e. by repeated application of col_join).

Examples

```
>>> from sympy import Matrix, eye
>>> Matrix.vstack(eye(2), 2*eye(2))
Matrix([
[1, 0],
[0, 1],
[2, 0],
[0, 2]])
```

classmethod wilkinson(*n*, ***kwargs*)

Returns two square Wilkinson Matrix of size 2*n + 1 $W_{2n+1}^-, W_{2n+1}^+ =$ Wilkinson(n)

Examples

```
>>> from sympy import Matrix
>>> wminus, wplus = Matrix.wilkinson(3)
>>> wminus
Matrix([
[-3,  1,  0, 0, 0, 0, 0],
[ 1, -2,  1, 0, 0, 0, 0],
[ 0,  1, -1, 1, 0, 0, 0],
[ 0,  0,  1, 0, 1, 0, 0],
[ 0,  0,  0, 1, 1, 1, 0],
[ 0,  0,  0, 0, 1, 2, 1],
[ 0,  0,  0, 0, 0, 1, 3]])
>>> wplus
Matrix([
[3, 1, 0, 0, 0, 0, 0],
[1, 2, 1, 0, 0, 0, 0],
[0, 1, 1, 1, 0, 0, 0],
[0, 0, 1, 0, 1, 0, 0],
[0, 0, 0, 1, 1, 1, 0],
[0, 0, 0, 0, 1, 2, 1],
[0, 0, 0, 0, 0, 1, 3]])
```

References

[R563], [R564]

xreplace(*rule*)

Return a new matrix with xreplace applied to each entry.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import SparseMatrix, Matrix
>>> SparseMatrix(1, 1, [x])
Matrix([[x]])
>>> _.xreplace({x: y})
Matrix([[y]])
>>> Matrix(_).xreplace({y: x})
Matrix([[x]])
```

classmethod zeros(*rows*, *cols=None*, ***kwargs*)

Returns a matrix of zeros.

Args

rows : rows of the matrix cols : cols of the matrix (if None, cols=rows)

Kwargs

cls : class of the returned matrix

MatrixKind

class sympy.matrices.common.**MatrixKind**(*element_kind=NumberKind*)

Kind for all matrices in SymPy.

Basic class for this kind is `MatrixBase` and `MatrixExpr`, but any expression representing the matrix can have this.

Parameters

element_kind : Kind

Kind of the element. Default is *sympy.core.kind.NumberKind* (page 1074), which means that the matrix contains only numbers.

Examples

Any instance of matrix class has `MatrixKind`:

```
>>> from sympy import MatrixSymbol
>>> A = MatrixSymbol('A', 2,2)
>>> A.kind
MatrixKind(NumberKind)
```

Although expression representing a matrix may be not instance of matrix class, it will have `MatrixKind` as well:

```
>>> from sympy import MatrixExpr, Integral
>>> from sympy.abc import x
>>> intM = Integral(A, x)
>>> isinstance(intM, MatrixExpr)
False
>>> intM.kind
MatrixKind(NumberKind)
```

Use `isinstance()` to check for `MatrixKind` without specifying the element kind. Use `is` with specifying the element kind:

```
>>> from sympy import Matrix
>>> from sympy.core import NumberKind
>>> from sympy.matrices import MatrixKind
>>> M = Matrix([1, 2])
>>> isinstance(M.kind, MatrixKind)
True
```

(continues on next page)

(continued from previous page)

```
>>> M.kind is MatrixKind(NumberKind)
True
```

See also:

sympy.core.kind.NumberKind (page 1074), *sympy.core.kind.UndefinedKind* (page 1074), *sympy.core.containers.TupleKind* (page 1069), *sympy.sets.sets.SetKind* (page 1217)

Dense Matrices

Matrix Class Reference

sympy.matrices.dense.**Matrix**

alias of *MutableDenseMatrix* (page 1363)

class sympy.matrices.dense.**DenseMatrix**

Matrix implementation based on DomainMatrix as the internal representation

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],
[ 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
```

(continues on next page)

(continued from previous page)

```
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)

as_immutable()

Returns an Immutable version of this Matrix

as_mutable()

Returns a mutable version of this matrix

Examples

```
>>> from sympy import ImmutableMatrix
>>> X = ImmutableMatrix([[1, 2], [3, 4]])
>>> Y = X.as_mutable()
>>> Y[1, 1] = 5 # Can set values in Y
>>> Y
Matrix([
[1, 2],
[3, 5]])
```

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
```

(continues on next page)

(continued from previous page)

```
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)

lower_triangular_solve(*rhs*)

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

See also:

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_solve* (page 1306)

upper_triangular_solve(*rhs*)

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

See also:

lower_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_solve* (page 1306)

class sympy.matrices.dense.**MutableDenseMatrix**(**args*, ***kwargs*)

simplify(***kwargs*)

Applies simplify to the elements of a matrix in place.

This is a shortcut for M.applyfunc(lambda x: simplify(x, ratio, measure))

See also:

sympy.simplify.simplify.simplify (page 661)

ImmutableMatrix Class Reference

class sympy.matrices.immutable.**ImmutableDenseMatrix**(**args*, ***kwargs*)

Create an immutable version of a matrix.

Examples

```
>>> from sympy import eye, ImmutableMatrix
>>> ImmutableMatrix(eye(3))
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
...
TypeError: Cannot set values of ImmutableDenseMatrix
```

Sparse Matrices

SparseMatrix Class Reference

sympy.matrices.sparse.**SparseMatrix**

alias of *MutableSparseMatrix* (page 1364)

class sympy.matrices.sparse.**MutableSparseMatrix**(**args*, ***kwargs*)

ImmutableSparseMatrix Class Reference

class sympy.matrices.immutable.**ImmutableSparseMatrix**(**args*, ***kwargs*)

Create an immutable version of a sparse matrix.

Examples

```
>>> from sympy import eye, ImmutableSparseMatrix
>>> ImmutableSparseMatrix(1, 1, {})
Matrix([[0]])
>>> ImmutableSparseMatrix(eye(3))
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
...
TypeError: Cannot set values of ImmutableSparseMatrix
>>> _.shape
(3, 3)
```

Sparse Tools

sympy.matrices.sparsetools.**_doktocsr**()

Converts a sparse matrix to Compressed Sparse Row (CSR) format.

Parameters

A : contains non-zero elements sorted by key (row, column)

JA : JA[i] is the column corresponding to A[i]

IA : IA[i] contains the index in A for the first non-zero element of row[i]. Thus IA[i+1] - IA[i] gives number of non-zero elements row[i]. The length of IA is always 1 more than the number of rows in the matrix.

Examples

```
>>> from sympy.matrices.sparsetools import _doktocsr
>>> from sympy import SparseMatrix, diag
>>> m = SparseMatrix(diag(1, 2, 3))
>>> m[2, 0] = -1
>>> _doktocsr(m)
[[1, 2, -1, 3], [0, 1, 0, 2], [0, 1, 2, 4], [3, 3]]
```

sympy.matrices.sparsetools.**_csrtodok**()

Converts a CSR representation to DOK representation.

Examples

```
>>> from sympy.matrices.sparsetools import _csrtodok
>>> _csrtodok([[5, 8, 3, 6], [0, 1, 2, 1], [0, 0, 2, 3, 4], [4, 3]])
Matrix([
[0, 0, 0],
[5, 8, 0],
[0, 0, 3],
[0, 6, 0]])
```

sympy.matrices.sparsetools.**banded**(***kwargs*)

Returns a SparseMatrix from the given dictionary describing the diagonals of the matrix. The keys are positive for upper diagonals and negative for those below the main diagonal. The values may be:

- expressions or single-argument functions,
- lists or tuples of values,
- matrices

Unless dimensions are given, the size of the returned matrix will be large enough to contain the largest non-zero value provided.

Kwargs

rows
: [rows of the resulting matrix; computed if] not given.

cols
: [columns of the resulting matrix; computed if] not given.

Examples

```
>>> from sympy import banded, ones, Matrix
>>> from sympy.abc import x
```

If explicit values are given in tuples, the matrix will autosize to contain all values, otherwise a single value is filled onto the entire diagonal:

```
>>> banded({1: (1, 2, 3), -1: (4, 5, 6), 0: x})
Matrix([
[x, 1, 0, 0],
[4, x, 2, 0],
[0, 5, x, 3],
[0, 0, 6, x]])
```

A function accepting a single argument can be used to fill the diagonal as a function of diagonal index (which starts at 0). The size (or shape) of the matrix must be given to obtain more than a 1x1 matrix:

```
>>> s = lambda d: (1 + d)**2
>>> banded(5, {0: s, 2: s, -2: 2})
Matrix([
[1, 0, 1,  0,  0],
[0, 4, 0,  4,  0],
[2, 0, 9,  0,  9],
[0, 2, 0, 16,  0],
[0, 0, 2,  0, 25]])
```

The diagonal of matrices placed on a diagonal will coincide with the indicated diagonal:

```
>>> vert = Matrix([1, 2, 3])
>>> banded({0: vert}, cols=3)
Matrix([
[1, 0, 0],
[2, 1, 0],
[3, 2, 1],
[0, 3, 2],
[0, 0, 3]])
```

```
>>> banded(4, {0: ones(2)})
Matrix([
[1, 1, 0, 0],
[1, 1, 0, 0],
[0, 0, 1, 1],
[0, 0, 1, 1]])
```

Errors are raised if the designated size will not hold all values an integral number of times. Here, the rows are designated as odd (but an even number is required to hold the off-diagonal 2x2 ones):

```
>>> banded({0: 2, 1: ones(2)}, rows=5)
Traceback (most recent call last):
...
ValueError:
sequence does not fit an integral number of times in the matrix
```

And here, an even number of rows is given...but the square matrix has an even number of columns, too. As we saw in the previous example, an odd number is required:

```
>>> banded(4, {0: 2, 1: ones(2)})  # trying to make 4x4 and cols must be␣
↪odd
Traceback (most recent call last):
...
ValueError:
sequence does not fit an integral number of times in the matrix
```

A way around having to count rows is to enclosing matrix elements in a tuple and indicate the desired number of them to the right:

```
>>> banded({0: 2, 2: (ones(2),)*3})
Matrix([
[2, 0, 1, 1, 0, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 2, 1, 1, 0, 0, 0, 0],
[0, 0, 2, 0, 1, 1, 0, 0],
[0, 0, 0, 2, 1, 1, 0, 0],
[0, 0, 0, 0, 2, 0, 1, 1],
[0, 0, 0, 0, 0, 2, 1, 1]])
```

An error will be raised if more than one value is written to a given entry. Here, the ones overlap with the main diagonal if they are placed on the first diagonal:

```
>>> banded({0: (2,)*5, 1: (ones(2),)*3})
Traceback (most recent call last):
...
ValueError: collision at (1, 1)
```

By placing a 0 at the bottom left of the 2x2 matrix of ones, the collision is avoided:

```
>>> u2 = Matrix([
... [1, 1],
... [0, 1]])
>>> banded({0: [2]*5, 1: [u2]*3})
Matrix([
[2, 1, 1, 0, 0, 0, 0],
[0, 2, 1, 0, 0, 0, 0],
[0, 0, 2, 1, 1, 0, 0],
[0, 0, 0, 2, 1, 0, 0],
[0, 0, 0, 0, 2, 1, 1],
[0, 0, 0, 0, 0, 0, 1]])
```


Immutable Matrices

The standard *Matrix* (page 1361) class in SymPy is mutable. This is important for performance reasons but means that standard matrices cannot interact well with the rest of SymPy. This is because the *Basic* (page 927) object, from which most SymPy classes inherit, is immutable.

The mission of the *ImmutableDenseMatrix* (page 1369) class, which is aliased as *ImmutableMatrix* (page 1369) for short, is to bridge the tension between performance/mutability and safety/immutability. Immutable matrices can do almost everything that normal matrices can do but they inherit from *Basic* (page 927) and can thus interact more naturally with the rest of SymPy. *ImmutableMatrix* (page 1369) also inherits from *MatrixExpr* (page 1370), allowing it to interact freely with SymPy's Matrix Expression module.

You can turn any Matrix-like object into an *ImmutableMatrix* (page 1369) by calling the constructor

```
>>> from sympy import Matrix, ImmutableMatrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M[1, 1] = 0
>>> IM = ImmutableMatrix(M)
>>> IM
Matrix([
[1, 2, 3],
```

(continues on next page)

(continued from previous page)

```
[4, 0, 6],
[7, 8, 9]])
>>> IM[1, 1] = 5
Traceback (most recent call last):
...
TypeError: Can not set values in Immutable Matrix. Use Matrix instead.
```

ImmutableMatrix Class Reference

sympy.matrices.immutable.**ImmutableMatrix**
alias of *ImmutableDenseMatrix* (page 1369)

class sympy.matrices.immutable.**ImmutableDenseMatrix**(**args*, ***kwargs*)
Create an immutable version of a matrix.

Examples

```
>>> from sympy import eye, ImmutableMatrix
>>> ImmutableMatrix(eye(3))
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
...
TypeError: Cannot set values of ImmutableDenseMatrix
```

Matrix Expressions

The Matrix expression module allows users to write down statements like

```
>>> from sympy import MatrixSymbol, Matrix
>>> X = MatrixSymbol('X', 3, 3)
>>> Y = MatrixSymbol('Y', 3, 3)
>>> (X.T*X).I*Y
X**(-1)*X.T**(-1)*Y
```

```
>>> Matrix(X)
Matrix([
[X[0, 0], X[0, 1], X[0, 2]],
[X[1, 0], X[1, 1], X[1, 2]],
[X[2, 0], X[2, 1], X[2, 2]]])
```

```
>>> (X*Y)[1, 2]
X[1, 0]*Y[0, 2] + X[1, 1]*Y[1, 2] + X[1, 2]*Y[2, 2]
```

where X and Y are *MatrixSymbol* (page 1372)'s rather than scalar symbols.

Matrix expression derivatives are supported. The derivative of a matrix by another matrix is generally a 4-dimensional array, but if some dimensions are trivial or diagonal, the derivation algorithm will try to express the result as a matrix expression:

```
>>> a = MatrixSymbol("a", 3, 1)
>>> b = MatrixSymbol("b", 3, 1)
>>> (a.T*X**2*b).diff(X)
a*b.T*X.T + X.T*a*b.T
```

```
>>> X.diff(X)
PermuteDims(ArrayTensorProduct(I, I), (3)(1 2))
```

The last output is an array expression, as the returned symbol is 4-dimensional.

Matrix Expressions Core Reference

class sympy.matrices.expressions.**MatrixExpr**(**args*, ***kwargs*)

Superclass for Matrix Expressions

MatrixExprs represent abstract matrices, linear transformations represented within a particular basis.

Examples

```
>>> from sympy import MatrixSymbol
>>> A = MatrixSymbol('A', 3, 3)
>>> y = MatrixSymbol('y', 3, 1)
>>> x = (A.T*A).I * A * y
```

See also:

MatrixSymbol (page 1372), *MatAdd* (page 1372), *MatMul* (page 1373), *Transpose* (page 1375), *Inverse* (page 1374)

property T

Matrix transposition

as_coeff_Mul(*rational=False*)

Efficiently extract the coefficient of a product.

as_explicit()

Returns a dense Matrix with elements represented explicitly

Returns an object of type ImmutableDenseMatrix.

Examples

```
>>> from sympy import Identity
>>> I = Identity(3)
>>> I
I
>>> I.as_explicit()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

See also:

as_mutable **(page 1371)**
returns mutable Matrix type

as_mutable()
Returns a dense, mutable matrix with elements represented explicitly

Examples

```
>>> from sympy import Identity
>>> I = Identity(3)
>>> I
I
>>> I.shape
(3, 3)
>>> I.as_mutable()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

See also:

as_explicit **(page 1370)**
returns ImmutableDenseMatrix

equals(*other*)
Test elementwise equality between matrices, potentially of different types

```
>>> from sympy import Identity, eye
>>> Identity(3).equals(eye(3))
True
```

static from_index_summation(*expr*, *first_index=None*, *last_index=None*, *dimensions=None*)
Parse expression of matrices with explicitly summed indices into a matrix expression without indices, if possible.

This transformation expressed in mathematical notation:

$\sum_{j=0}^{N-1} A_{i,j} B_{j,k} \Longrightarrow \mathbf{A} \cdot \mathbf{B}$

Optional parameter `first_index`: specify which free index to use as the index starting the expression.

Examples

```
>>> from sympy import MatrixSymbol, MatrixExpr, Sum
>>> from sympy.abc import i, j, k, l, N
>>> A = MatrixSymbol("A", N, N)
>>> B = MatrixSymbol("B", N, N)
>>> expr = Sum(A[i, j]*B[j, k], (j, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
A*B
```

Transposition is detected:

```
>>> expr = Sum(A[j, i]*B[j, k], (j, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
A.T*B
```

Detect the trace:

```
>>> expr = Sum(A[i, i], (i, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
Trace(A)
```

More complicated expressions:

```
>>> expr = Sum(A[i, j]*B[k, j]*A[l, k], (j, 0, N-1), (k, 0, N-1))
>>> MatrixExpr.from_index_summation(expr)
A*B.T*A.T
```

class sympy.matrices.expressions.**MatrixSymbol**(*name*, *n*, *m*)

Symbolic representation of a Matrix object

Creates a SymPy Symbol to represent a Matrix. This matrix has a shape and can be included in Matrix Expressions

Examples

```
>>> from sympy import MatrixSymbol, Identity
>>> A = MatrixSymbol('A', 3, 4) # A 3 by 4 Matrix
>>> B = MatrixSymbol('B', 4, 3) # A 4 by 3 Matrix
>>> A.shape
(3, 4)
>>> 2*A*B + Identity(3)
I + 2*A*B
```

class sympy.matrices.expressions.**MatAdd**(**args*, *evaluate=False*, *check=None*, *_sympify=True*)

A Sum of Matrix Expressions

MatAdd inherits from and operates like SymPy Add

Examples

```
>>> from sympy import MatAdd, MatrixSymbol
>>> A = MatrixSymbol('A', 5, 5)
>>> B = MatrixSymbol('B', 5, 5)
>>> C = MatrixSymbol('C', 5, 5)
>>> MatAdd(A, B, C)
A + B + C
```

class sympy.matrices.expressions.**MatMul**(**args*, *evaluate=False*, *check=None*, *_sympify=True*)

A product of matrix expressions

Examples

```
>>> from sympy import MatMul, MatrixSymbol
>>> A = MatrixSymbol('A', 5, 4)
>>> B = MatrixSymbol('B', 4, 3)
>>> C = MatrixSymbol('C', 3, 6)
>>> MatMul(A, B, C)
A*B*C
```

class sympy.matrices.expressions.**MatPow**(*base*, *exp*, *evaluate=False*, ***options*)

class sympy.matrices.expressions.**HadamardProduct**(**args*, *evaluate=False*, *check=None*)

Elementwise product of matrix expressions

Examples

Hadamard product for matrix symbols:

```
>>> from sympy import hadamard_product, HadamardProduct, MatrixSymbol
>>> A = MatrixSymbol('A', 5, 5)
>>> B = MatrixSymbol('B', 5, 5)
>>> isinstance(hadamard_product(A, B), HadamardProduct)
True
```

Notes

This is a symbolic object that simply stores its argument without evaluating it. To actually compute the product, use the function `hadamard_product()` or `HadamardProduct.doit`

class sympy.matrices.expressions.**HadamardPower**(*base*, *exp*)

Elementwise power of matrix expressions

Parameters

base : scalar or matrix

exp : scalar or matrix

Notes

There are four definitions for the hadamard power which can be used. Let's consider A, B as (m, n) matrices, and a, b as scalars.

Matrix raised to a scalar exponent:

$$A^{\circ b} = \begin{bmatrix} A_{0,0}^b & A_{0,1}^b & \cdots & A_{0,n-1}^b \\ A_{1,0}^b & A_{1,1}^b & \cdots & A_{1,n-1}^b \\ \vdots & \vdots & \ddots & \vdots \\ A_{m-1,0}^b & A_{m-1,1}^b & \cdots & A_{m-1,n-1}^b \end{bmatrix}$$

Scalar raised to a matrix exponent:

$$a^{\circ B} = \begin{bmatrix} a^{B_{0,0}} & a^{B_{0,1}} & \cdots & a^{B_{0,n-1}} \\ a^{B_{1,0}} & a^{B_{1,1}} & \cdots & a^{B_{1,n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ a^{B_{m-1,0}} & a^{B_{m-1,1}} & \cdots & a^{B_{m-1,n-1}} \end{bmatrix}$$

Matrix raised to a matrix exponent:

$$A^{\circ B} = \begin{bmatrix} A_{0,0}^{B_{0,0}} & A_{0,1}^{B_{0,1}} & \cdots & A_{0,n-1}^{B_{0,n-1}} \\ A_{1,0}^{B_{1,0}} & A_{1,1}^{B_{1,1}} & \cdots & A_{1,n-1}^{B_{1,n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m-1,0}^{B_{m-1,0}} & A_{m-1,1}^{B_{m-1,1}} & \cdots & A_{m-1,n-1}^{B_{m-1,n-1}} \end{bmatrix}$$

Scalar raised to a scalar exponent:

$$a^{\circ b} = a^b$$

class sympy.matrices.expressions.**Inverse**(*mat*, *exp=-1*)

The multiplicative inverse of a matrix expression

This is a symbolic object that simply stores its argument without evaluating it. To actually compute the inverse, use the `.inverse()` method of matrices.

Examples

```
>>> from sympy import MatrixSymbol, Inverse
>>> A = MatrixSymbol('A', 3, 3)
>>> B = MatrixSymbol('B', 3, 3)
>>> Inverse(A)
A**(-1)
>>> A.inverse() == Inverse(A)
True
>>> (A*B).inverse()
B**(-1)*A**(-1)
>>> Inverse(A*B)
(A*B)**(-1)
```

class sympy.matrices.expressions.**Transpose**(**args*, ***kwargs*)

The transpose of a matrix expression.

This is a symbolic object that simply stores its argument without evaluating it. To actually compute the transpose, use the `transpose()` function, or the `.T` attribute of matrices.

Examples

```
>>> from sympy import MatrixSymbol, Transpose, transpose
>>> A = MatrixSymbol('A', 3, 5)
>>> B = MatrixSymbol('B', 5, 3)
>>> Transpose(A)
A.T
>>> A.T == transpose(A) == Transpose(A)
True
>>> Transpose(A*B)
(A*B).T
>>> transpose(A*B)
B.T*A.T
```

class sympy.matrices.expressions.**Trace**(*mat*)

Matrix Trace

Represents the trace of a matrix expression.

Examples

```
>>> from sympy import MatrixSymbol, Trace, eye
>>> A = MatrixSymbol('A', 3, 3)
>>> Trace(A)
Trace(A)
>>> Trace(eye(3))
Trace(Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]]))
```

(continues on next page)

(continued from previous page)

```
>>> Trace(eye(3)).simplify()
3
```

class sympy.matrices.expressions.**FunctionMatrix**(*rows*, *cols*, *lamda*)

Represents a matrix using a function (`Lambda`) which gives outputs according to the coordinates of each matrix entries.

Parameters

rows : nonnegative integer. Can be symbolic.

cols : nonnegative integer. Can be symbolic.

lamda : Function, Lambda or str

If it is a SymPy `Function` or `Lambda` instance, it should be able to accept two arguments which represents the matrix coordinates.

If it is a pure string containing Python `lambda` semantics, it is interpreted by the SymPy parser and casted into a SymPy `Lambda` instance.

Examples

Creating a `FunctionMatrix` from `Lambda`:

```
>>> from sympy import FunctionMatrix, symbols, Lambda, MatPow
>>> i, j, n, m = symbols('i,j,n,m')
>>> FunctionMatrix(n, m, Lambda((i, j), i + j))
FunctionMatrix(n, m, Lambda((i, j), i + j))
```

Creating a `FunctionMatrix` from a SymPy function:

```
>>> from sympy import KroneckerDelta
>>> X = FunctionMatrix(3, 3, KroneckerDelta)
>>> X.as_explicit()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

Creating a `FunctionMatrix` from a SymPy undefined function:

```
>>> from sympy import Function
>>> f = Function('f')
>>> X = FunctionMatrix(3, 3, f)
>>> X.as_explicit()
Matrix([
[f(0, 0), f(0, 1), f(0, 2)],
[f(1, 0), f(1, 1), f(1, 2)],
[f(2, 0), f(2, 1), f(2, 2)]])
```

Creating a `FunctionMatrix` from Python `lambda`:

```
>>> FunctionMatrix(n, m, 'lambda i, j: i + j')
FunctionMatrix(n, m, Lambda((i, j), i + j))
```

Example of lazy evaluation of matrix product:

```
>>> Y = FunctionMatrix(1000, 1000, Lambda((i, j), i + j))
>>> isinstance(Y*Y, MatPow) # this is an expression object
True
>>> (Y**2)[10,10] # So this is evaluated lazily
342923500
```

Notes

This class provides an alternative way to represent an extremely dense matrix with entries in some form of a sequence, in a most sparse way.

class sympy.matrices.expressions.**PermutationMatrix**(*perm*)

A Permutation Matrix

Parameters

perm : Permutation

The permutation the matrix uses.

The size of the permutation determines the matrix size.

See the documentation of *sympy.combinatorics.permutations.Permutation* (page 257) for the further information of how to create a permutation object.

Examples

```
>>> from sympy import Matrix, PermutationMatrix
>>> from sympy.combinatorics import Permutation
```

Creating a permutation matrix:

```
>>> p = Permutation(1, 2, 0)
>>> P = PermutationMatrix(p)
>>> P = P.as_explicit()
>>> P
Matrix([
[0, 1, 0],
[0, 0, 1],
[1, 0, 0]])
```

Permuting a matrix row and column:

```
>>> M = Matrix([0, 1, 2])
>>> Matrix(P*M)
Matrix([
[1],
[2],
[0]])
```

```
>>> Matrix(M.T*P)
Matrix([[2, 0, 1]])
```

See also:

sympy.combinatorics.permutations.Permutation (page 257)

class sympy.matrices.expressions.**MatrixPermute**(*mat*, *perm*, *axis=0*)

Symbolic representation for permuting matrix rows or columns.

Parameters

perm : Permutation, PermutationMatrix

The permutation to use for permuting the matrix. The permutation can be resized to the suitable one,

axis : 0 or 1

The axis to permute alongside. If 0, it will permute the matrix rows. If 1, it will permute the matrix columns.

Notes

This follows the same notation used in *sympy.matrices.common.MatrixCommon.permute()* (page 1348).

Examples

```
>>> from sympy import Matrix, MatrixPermute
>>> from sympy.combinatorics import Permutation
```

Permuting the matrix rows:

```
>>> p = Permutation(1, 2, 0)
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> B = MatrixPermute(A, p, axis=0)
>>> B.as_explicit()
Matrix([
[4, 5, 6],
[7, 8, 9],
[1, 2, 3]])
```

Permuting the matrix columns:

```
>>> B = MatrixPermute(A, p, axis=1)
>>> B.as_explicit()
Matrix([
[2, 3, 1],
[5, 6, 4],
[8, 9, 7]])
```

See also:

sympy.matrices.common.MatrixCommon.permute (page 1348)

class sympy.matrices.expressions.**Identity**(*n*)

The Matrix Identity I - multiplicative identity

Examples

```
>>> from sympy import Identity, MatrixSymbol
>>> A = MatrixSymbol('A', 3, 5)
>>> I = Identity(3)
>>> I*A
A
```

class sympy.matrices.expressions.**ZeroMatrix**(*m*, *n*)

The Matrix Zero 0 - additive identity

Examples

```
>>> from sympy import MatrixSymbol, ZeroMatrix
>>> A = MatrixSymbol('A', 3, 5)
>>> Z = ZeroMatrix(3, 5)
>>> A + Z
A
>>> Z*A.T
0
```

class sympy.matrices.expressions.**CompanionMatrix**(*poly*)

A symbolic companion matrix of a polynomial.

Examples

```
>>> from sympy import Poly, Symbol, symbols
>>> from sympy.matrices.expressions import CompanionMatrix
>>> x = Symbol('x')
>>> c0, c1, c2, c3, c4 = symbols('c0:5')
>>> p = Poly(c0 + c1*x + c2*x**2 + c3*x**3 + c4*x**4 + x**5, x)
>>> CompanionMatrix(p)
CompanionMatrix(Poly(x**5 + c4*x**4 + c3*x**3 + c2*x**2 + c1*x + c0,
x, domain='ZZ[c0,c1,c2,c3,c4]'))
```

class sympy.matrices.expressions.**MatrixSet**(*n*, *m*, *set*)

MatrixSet represents the set of matrices with `shape = (n, m)` over the given set.

Examples

```
>>> from sympy.matrices import MatrixSet
>>> from sympy import S, I, Matrix
>>> M = MatrixSet(2, 2, set=S.Reals)
>>> X = Matrix([[1, 2], [3, 4]])
>>> X in M
True
>>> X = Matrix([[1, 2], [I, 4]])
>>> X in M
False
```

Block Matrices

Block matrices allow you to construct larger matrices out of smaller sub-blocks. They can work with *MatrixExpr* (page 1370) or *ImmutableMatrix* (page 1369) objects.

class sympy.matrices.expressions.blockmatrix.**BlockMatrix**(**args*, ***kwargs*)

A BlockMatrix is a Matrix comprised of other matrices.

The submatrices are stored in a SymPy Matrix object but accessed as part of a Matrix Expression

```
>>> from sympy import (MatrixSymbol, BlockMatrix, symbols,
...     Identity, ZeroMatrix, block_collapse)
>>> n,m,l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m,n), Y]])
>>> print(B)
Matrix([
[X, Z],
[0, Y]])
```

```
>>> C = BlockMatrix([[Identity(n), Z]])
>>> print(C)
Matrix([[I, Z]])
```

```
>>> print(block_collapse(C*B))
Matrix([[X, Z + Z*Y]])
```

Some matrices might be comprised of rows of blocks with the matrices in each row having the same height and the rows all having the same total number of columns but not having the same number of columns for each matrix in each row. In this case, the matrix is not a block matrix and should be instantiated by Matrix.

```
>>> from sympy import ones, Matrix
>>> dat = [
... [ones(3,2), ones(3,3)*2],
... [ones(2,3)*3, ones(2,2)*4]]
...
```

(continues on next page)

(continued from previous page)

```
>>> BlockMatrix(dat)
Traceback (most recent call last):
...
ValueError:
Although this matrix is comprised of blocks, the blocks do not fill
the matrix in a size-symmetric fashion. To create a full matrix from
these arguments, pass them directly to Matrix.
>>> Matrix(dat)
Matrix([
[1, 1, 2, 2, 2],
[1, 1, 2, 2, 2],
[1, 1, 2, 2, 2],
[3, 3, 3, 4, 4],
[3, 3, 3, 4, 4]])
```

See also:

sympy.matrices.matrices.MatrixBase.irregular (page 1302)

LDUdecomposition()

Returns the Block LDU decomposition of a 2x2 Block Matrix

Returns

(L, D, U) : Matrices

L : Lower Diagonal Matrix D : Diagonal Matrix U : Upper Diagonal Matrix

Raises

ShapeError

If the block matrix is not a 2x2 matrix

NonInvertibleMatrixError

If the matrix “A” is non-invertible

Examples

```
>>> from sympy import symbols, MatrixSymbol, BlockMatrix, block_
↪collapse
>>> m, n = symbols('m n')
>>> A = MatrixSymbol('A', n, n)
>>> B = MatrixSymbol('B', n, m)
>>> C = MatrixSymbol('C', m, n)
>>> D = MatrixSymbol('D', m, m)
>>> X = BlockMatrix([[A, B], [C, D]])
>>> L, D, U = X.LDUdecomposition()
>>> block_collapse(L*D*U)
Matrix([
[A, B],
[C, D]])
```

See also:

sympy.matrices.expressions.blockmatrix.BlockMatrix.UDLdecomposition (page 1382), *sympy.matrices.expressions.blockmatrix.BlockMatrix.LUdecomposition* (page 1382)

LUdecomposition()

Returns the Block LU decomposition of a 2x2 Block Matrix

Returns

(L, U) : Matrices

L : Lower Diagonal Matrix U : Upper Diagonal Matrix

Raises

ShapeError

If the block matrix is not a 2x2 matrix

NonInvertibleMatrixError

If the matrix "A" is non-invertible

Examples

```
>>> from sympy import symbols, MatrixSymbol, BlockMatrix, block_
↪collapse
>>> m, n = symbols('m n')
>>> A = MatrixSymbol('A', n, n)
>>> B = MatrixSymbol('B', n, m)
>>> C = MatrixSymbol('C', m, n)
>>> D = MatrixSymbol('D', m, m)
>>> X = BlockMatrix([[A, B], [C, D]])
>>> L, U = X.LUdecomposition()
>>> block_collapse(L*U)
Matrix([
[A, B],
[C, D]])
```

See also:

sympy.matrices.expressions.blockmatrix.BlockMatrix.UDLdecomposition (page 1382), *sympy.matrices.expressions.blockmatrix.BlockMatrix.LDUdecomposition* (page 1381)

UDLdecomposition()

Returns the Block UDL decomposition of a 2x2 Block Matrix

Returns

(U, D, L) : Matrices

U : Upper Diagonal Matrix D : Diagonal Matrix L : Lower Diagonal Matrix

Raises

ShapeError

If the block matrix is not a 2x2 matrix

NonInvertibleMatrixError

If the matrix “D” is non-invertible

Examples

```
>>> from sympy import symbols, MatrixSymbol, BlockMatrix, block_
↪collapse
>>> m, n = symbols('m n')
>>> A = MatrixSymbol('A', n, n)
>>> B = MatrixSymbol('B', n, m)
>>> C = MatrixSymbol('C', m, n)
>>> D = MatrixSymbol('D', m, m)
>>> X = BlockMatrix([[A, B], [C, D]])
>>> U, D, L = X.UDLdecomposition()
>>> block_collapse(U*D*L)
Matrix([
[A, B],
[C, D]])
```

See also:

sympy.matrices.expressions.blockmatrix.BlockMatrix.LDUdecomposition (page 1381), *sympy.matrices.expressions.blockmatrix.BlockMatrix.LUdecomposition* (page 1382)

schur(*mat='A'*, *generalized=False*)

Return the Schur Complement of the 2x2 BlockMatrix

Parameters

mat : String, optional

The matrix with respect to which the Schur Complement is calculated. ‘A’ is used by default

generalized : bool, optional

If True, returns the generalized Schur Component which uses Moore-Penrose Inverse

Returns

M : Matrix

The Schur Complement Matrix

Raises

ShapeError

If the block matrix is not a 2x2 matrix

NonInvertibleMatrixError

If given matrix is non-invertible

Examples

```
>>> from sympy import symbols, MatrixSymbol, BlockMatrix
>>> m, n = symbols('m n')
>>> A = MatrixSymbol('A', n, n)
>>> B = MatrixSymbol('B', n, m)
>>> C = MatrixSymbol('C', m, n)
>>> D = MatrixSymbol('D', m, m)
>>> X = BlockMatrix([[A, B], [C, D]])
```

The default Schur Complement is evaluated with "A"

```
>>> X.schur()
-C*A**(-1)*B + D
>>> X.schur('D')
A - B*D**(-1)*C
```

Schur complement with non-invertible matrices is not defined. Instead, the generalized Schur complement can be calculated which uses the Moore-Penrose Inverse. To achieve this, *generalized* must be set to *True*

```
>>> X.schur('B', generalized=True)
C - D*(B.T*B)**(-1)*B.T*A
>>> X.schur('C', generalized=True)
-A*(C.T*C)**(-1)*C.T*D + B
```

See also:

sympy.matrices.matrices.MatrixBase.pinv (page 1305)

References

[R565]

transpose()

Return transpose of matrix.

Examples

```
>>> from sympy import MatrixSymbol, BlockMatrix, ZeroMatrix
>>> from sympy.abc import m, n
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m,n), Y]])
>>> B.transpose()
Matrix([
[X.T,  0],
[Z.T, Y.T]])
>>> _.transpose()
Matrix([
```

(continues on next page)

(continued from previous page)

```
[X, Z],
[0, Y]])
```

class sympy.matrices.expressions.blockmatrix.**BlockDiagMatrix**(**mats*)

A sparse matrix with block matrices along its diagonals

Examples

```
>>> from sympy import MatrixSymbol, BlockDiagMatrix, symbols
>>> n, m, l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> BlockDiagMatrix(X, Y)
Matrix([
[X, 0],
[0, Y]])
```

Notes

If you want to get the individual diagonal blocks, use *get_diag_blocks()* (page 1385).

See also:

sympy.matrices.dense.diag (page 1319)

get_diag_blocks()

Return the list of diagonal blocks of the matrix.

Examples

```
>>> from sympy import BlockDiagMatrix, Matrix
```

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> B = Matrix([[5, 6], [7, 8]])
>>> M = BlockDiagMatrix(A, B)
```

How to get diagonal blocks from the block diagonal matrix:

```
>>> diag_blocks = M.get_diag_blocks()
>>> diag_blocks[0]
Matrix([
[1, 2],
[3, 4]])
>>> diag_blocks[1]
Matrix([
[5, 6],
[7, 8]])
```

sympy.matrices.expressions.blockmatrix.**block_collapse**(*expr*)

Evaluates a block matrix expression

```
>>> from sympy import MatrixSymbol, BlockMatrix, symbols, Identity,␣
↪ZeroMatrix, block_collapse
>>> n,m,l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m, m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])
>>> print(B)
Matrix([
[X, Z],
[0, Y]])
```

```
>>> C = BlockMatrix([[Identity(n), Z]])
>>> print(C)
Matrix([[I, Z]])
```

```
>>> print(block_collapse(C*B))
Matrix([[X, Z + Z*Y]])
```

Matrix Normal Forms

sympy.matrices.normalforms.**smith_normal_form**(*m*, *domain=None*)

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

Examples

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

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

Compute the Hermite Normal Form of a Matrix *A* of integers.

Parameters

A : $m \times n$ `Matrix` of integers.

D : int, optional

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

check_rank : boolean, optional (default=False)

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

Returns

`Matrix`

The HNF of matrix *A*.

Raises

DMDomainError

If the domain of the matrix is not *ZZ* (page 2525).

DMShapeError

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

Examples

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

References

[R598]

Tensor

A module to manipulate symbolic objects with indices including tensors

Contents

N-dim array

N-dim array module for SymPy.

Four classes are provided to handle N-dim arrays, given by the combinations dense/sparse (i.e. whether to store all elements or only the non-zero ones in memory) and mutable/immutable (immutable classes are SymPy objects, but cannot change after they have been created).

Examples

The following examples show the usage of `Array`. This is an abbreviation for `ImmutableDenseNDimArray`, that is an immutable and dense N-dim array, the other classes are analogous. For mutable classes it is also possible to change element values after the object has been constructed.

Array construction can detect the shape of nested lists and tuples:

```
>>> from sympy import Array
>>> a1 = Array([[1, 2], [3, 4], [5, 6]])
>>> a1
[[1, 2], [3, 4], [5, 6]]
>>> a1.shape
(3, 2)
>>> a1.rank()
2
>>> from sympy.abc import x, y, z
>>> a2 = Array([[[x, y], [z, x*z]], [[1, x*y], [1/x, x/y]]])
>>> a2
[[[x, y], [z, x*z]], [[1, x*y], [1/x, x/y]]]
>>> a2.shape
(2, 2, 2)
>>> a2.rank()
3
```

Otherwise one could pass a 1-dim array followed by a shape tuple:

```
>>> m1 = Array(range(12), (3, 4))
>>> m1
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
>>> m2 = Array(range(12), (3, 2, 2))
>>> m2
[[[0, 1], [2, 3]], [[4, 5], [6, 7]], [[8, 9], [10, 11]]]
>>> m2[1,1,1]
7
>>> m2.reshape(4, 3)
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]]
```

Slice support:

```
>>> m2[:, 1, 1]
[3, 7, 11]
```

Elementwise derivative:

```
>>> from sympy.abc import x, y, z
>>> m3 = Array([x**3, x*y, z])
>>> m3.diff(x)
[3*x**2, y, 0]
>>> m3.diff(z)
[0, 0, 1]
```

Multiplication with other SymPy expressions is applied elementwisely:

```
>>> (1+x)*m3
[x**3*(x + 1), x*y*(x + 1), z*(x + 1)]
```

To apply a function to each element of the N-dim array, use `applyfunc`:

```
>>> m3.applyfunc(lambda x: x/2)
[x**3/2, x*y/2, z/2]
```

N-dim arrays can be converted to nested lists by the `tolist()` method:

```
>>> m2.tolist()
[[[0, 1], [2, 3]], [[4, 5], [6, 7]], [[8, 9], [10, 11]]]
>>> isinstance(m2.tolist(), list)
True
```

If the rank is 2, it is possible to convert them to matrices with `tomatrix()`:

```
>>> m1.tomatrix()
Matrix([
[0, 1,  2,  3],
[4, 5,  6,  7],
[8, 9, 10, 11]])
```

Products and contractions

Tensor product between arrays $A_{i_1,\ldots,i_n}$ and $B_{j_1,\ldots,j_m}$ creates the combined array $P = A \otimes B$ defined as

$P_{i_1,\ldots,i_n,j_1,\ldots,j_m} := A_{i_1,\ldots,i_n} \cdot B_{j_1,\ldots,j_m}$.

It is available through `tensorproduct(...)`:

```
>>> from sympy import Array, tensorproduct
>>> from sympy.abc import x,y,z,t
>>> A = Array([x, y, z, t])
>>> B = Array([1, 2, 3, 4])
>>> tensorproduct(A, B)
[[x, 2*x, 3*x, 4*x], [y, 2*y, 3*y, 4*y], [z, 2*z, 3*z, 4*z], [t, 2*t, 3*t,␣
↪4*t]]
```

Tensor product between a rank-1 array and a matrix creates a rank-3 array:

```
>>> from sympy import eye
>>> p1 = tensorproduct(A, eye(4))
>>> p1
[[[x, 0, 0, 0], [0, x, 0, 0], [0, 0, x, 0], [0, 0, 0, x]], [[y, 0, 0, 0], [0,␣
↪y, 0, 0], [0, 0, y, 0], [0, 0, 0, y]], [[z, 0, 0, 0], [0, z, 0, 0], [0, 0,␣
↪z, 0], [0, 0, 0, z]], [[t, 0, 0, 0], [0, t, 0, 0], [0, 0, t, 0], [0, 0, 0,␣
↪t]]]
```

Now, to get back $A_0 \otimes \mathbf{1}$ one can access $p_{0,m,n}$ by slicing:

```
>>> p1[0,:,:]
[[x, 0, 0, 0], [0, x, 0, 0], [0, 0, x, 0], [0, 0, 0, x]]
```

Tensor contraction sums over the specified axes, for example contracting positions a and b means

$A_{i_1,\ldots,i_a,\ldots,i_b,\ldots,i_n} \implies \sum_k A_{i_1,\ldots,k,\ldots,k,\ldots,i_n}$

Remember that Python indexing is zero starting, to contract the a-th and b-th axes it is therefore necessary to specify $a-1$ and $b-1$

```
>>> from sympy import tensorcontraction
>>> C = Array([[x, y], [z, t]])
```

The matrix trace is equivalent to the contraction of a rank-2 array:

$A_{m,n} \implies \sum_k A_{k,k}$

```
>>> tensorcontraction(C, (0, 1))
t + x
```

Matrix product is equivalent to a tensor product of two rank-2 arrays, followed by a contraction of the 2nd and 3rd axes (in Python indexing axes number 1, 2).

$A_{m,n} \cdot B_{i,j} \implies \sum_k A_{m,k} \cdot B_{k,j}$

```
>>> D = Array([[2, 1], [0, -1]])
>>> tensorcontraction(tensorproduct(C, D), (1, 2))
[[2*x, x - y], [2*z, -t + z]]
```

One may verify that the matrix product is equivalent:

```
>>> from sympy import Matrix
>>> Matrix([[x, y], [z, t]])*Matrix([[2, 1], [0, -1]])
Matrix([
[2*x,  x - y],
[2*z, -t + z]])
```

or equivalently

```
>>> C.tomatrix()*D.tomatrix()
Matrix([
[2*x,  x - y],
[2*z, -t + z]])
```

Diagonal operator

The `tensordiagonal` function acts in a similar manner as `tensorcontraction`, but the joined indices are not summed over, for example diagonalizing positions a and b means

$A_{i_1,\ldots,i_a,\ldots,i_b,\ldots,i_n} \implies A_{i_1,\ldots,k,\ldots,k,\ldots,i_n} \implies \tilde{A}_{i_1,\ldots,i_{a-1},i_{a+1},\ldots,i_{b-1},i_{b+1},\ldots,i_n,k}$

where $\tilde{A}$ is the array equivalent to the diagonal of A at positions a and b moved to the last index slot.

Compare the difference between contraction and diagonal operators:

```
>>> from sympy import tensordiagonal
>>> from sympy.abc import a, b, c, d
>>> m = Matrix([[a, b], [c, d]])
>>> tensorcontraction(m, [0, 1])
a + d
>>> tensordiagonal(m, [0, 1])
[a, d]
```

In short, no summation occurs with `tensordiagonal`.

Derivatives by array

The usual derivative operation may be extended to support derivation with respect to arrays, provided that all elements in the that array are symbols or expressions suitable for derivations.

The definition of a derivative by an array is as follows: given the array $A_{i_1,\ldots,i_N}$ and the array $X_{j_1,\ldots,j_M}$ the derivative of arrays will return a new array B defined by

$B_{j_1,\ldots,j_M,i_1,\ldots,i_N} := \frac{\partial A_{i_1,\ldots,i_N}}{\partial X_{j_1,\ldots,j_M}}$

The function `derive_by_array` performs such an operation:

```
>>> from sympy import derive_by_array
>>> from sympy.abc import x, y, z, t
>>> from sympy import sin, exp
```

With scalars, it behaves exactly as the ordinary derivative:

```
>>> derive_by_array(sin(x*y), x)
y*cos(x*y)
```

Scalar derived by an array basis:

```
>>> derive_by_array(sin(x*y), [x, y, z])
[y*cos(x*y), x*cos(x*y), 0]
```

Deriving array by an array basis: $B^{nm} := \frac{\partial A^m}{\partial x^n}$

```
>>> basis = [x, y, z]
>>> ax = derive_by_array([exp(x), sin(y*z), t], basis)
>>> ax
[[exp(x), 0, 0], [0, z*cos(y*z), 0], [0, y*cos(y*z), 0]]
```

Contraction of the resulting array: $\sum_m \frac{\partial A^m}{\partial x^m}$

```
>>> tensorcontraction(ax, (0, 1))
z*cos(y*z) + exp(x)
```

Classes

class sympy.tensor.array.**ImmutableDenseNDimArray**(*iterable*, *shape=None*, ***kwargs*)

class sympy.tensor.array.**ImmutableSparseNDimArray**(*iterable=None*, *shape=None*, ***kwargs*)

class sympy.tensor.array.**MutableDenseNDimArray**(*iterable=None*, *shape=None*, ***kwargs*)

class sympy.tensor.array.**MutableSparseNDimArray**(*iterable=None*, *shape=None*, ***kwargs*)

Functions

sympy.tensor.array.**derive_by_array**(*expr*, *dx*)

Derivative by arrays. Supports both arrays and scalars.

Explanation

Given the array $A_{i_1,\ldots,i_N}$ and the array $X_{j_1,\ldots,j_M}$ this function will return a new array B defined by

$B_{j_1,\ldots,j_M,i_1,\ldots,i_N} := \frac{\partial A_{i_1,\ldots,i_N}}{\partial X_{j_1,\ldots,j_M}}$

Examples

```
>>> from sympy import derive_by_array
>>> from sympy.abc import x, y, z, t
>>> from sympy import cos
>>> derive_by_array(cos(x*t), x)
-t*sin(t*x)
>>> derive_by_array(cos(x*t), [x, y, z, t])
[-t*sin(t*x), 0, 0, -x*sin(t*x)]
>>> derive_by_array([x, y**2*z], [[x, y], [z, t]])
[[[1, 0], [0, 2*y*z]], [[0, y**2], [0, 0]]]
```

sympy.tensor.array.**permutedims**(*expr*, *perm=None*, *index_order_old=None*, *index_order_new=None*)

Permutes the indices of an array.

Parameter specifies the permutation of the indices.

Examples

```
>>> from sympy.abc import x, y, z, t
>>> from sympy import sin
>>> from sympy import Array, permutedims
>>> a = Array([[x, y, z], [t, sin(x), 0]])
>>> a
[[x, y, z], [t, sin(x), 0]]
>>> permutedims(a, (1, 0))
[[x, t], [y, sin(x)], [z, 0]]
```

If the array is of second order, `transpose` can be used:

```
>>> from sympy import transpose
>>> transpose(a)
[[x, t], [y, sin(x)], [z, 0]]
```

Examples on higher dimensions:

```
>>> b = Array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
>>> permutedims(b, (2, 1, 0))
[[[1, 5], [3, 7]], [[2, 6], [4, 8]]]
>>> permutedims(b, (1, 2, 0))
[[[1, 5], [2, 6]], [[3, 7], [4, 8]]]
```

An alternative way to specify the same permutations as in the previous lines involves passing the *old* and *new* indices, either as a list or as a string:

```
>>> permutedims(b, index_order_old="cba", index_order_new="abc")
[[[1, 5], [3, 7]], [[2, 6], [4, 8]]]
>>> permutedims(b, index_order_old="cab", index_order_new="abc")
[[[1, 5], [2, 6]], [[3, 7], [4, 8]]]
```

`Permutation` objects are also allowed:

```
>>> from sympy.combinatorics import Permutation
>>> permutedims(b, Permutation([1, 2, 0]))
[[[1, 5], [2, 6]], [[3, 7], [4, 8]]]
```

sympy.tensor.array.**tensorcontraction**(*array*, **contraction_axes*)

Contraction of an array-like object on the specified axes.

Examples

```
>>> from sympy import Array, tensorcontraction
>>> from sympy import Matrix, eye
>>> tensorcontraction(eye(3), (0, 1))
3
>>> A = Array(range(18), (3, 2, 3))
>>> A
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]], [[12, 13, 14], [15,␣
↪16, 17]]]
```

(continues on next page)

(continued from previous page)

```
>>> tensorcontraction(A, (0, 2))
[21, 30]
```

Matrix multiplication may be emulated with a proper combination of `tensorcontraction` and `tensorproduct`

```
>>> from sympy import tensorproduct
>>> from sympy.abc import a,b,c,d,e,f,g,h
>>> m1 = Matrix([[a, b], [c, d]])
>>> m2 = Matrix([[e, f], [g, h]])
>>> p = tensorproduct(m1, m2)
>>> p
[[[[a*e, a*f], [a*g, a*h]], [[b*e, b*f], [b*g, b*h]]], [[[c*e, c*f],␣
↪[c*g, c*h]], [[d*e, d*f], [d*g, d*h]]]]
>>> tensorcontraction(p, (1, 2))
[[a*e + b*g, a*f + b*h], [c*e + d*g, c*f + d*h]]
>>> m1*m2
Matrix([
[a*e + b*g, a*f + b*h],
[c*e + d*g, c*f + d*h]])
```

sympy.tensor.array.**tensorproduct**(**args*)

Tensor product among scalars or array-like objects.

Examples

```
>>> from sympy.tensor.array import tensorproduct, Array
>>> from sympy.abc import x, y, z, t
>>> A = Array([[1, 2], [3, 4]])
>>> B = Array([x, y])
>>> tensorproduct(A, B)
[[[x, y], [2*x, 2*y]], [[3*x, 3*y], [4*x, 4*y]]]
>>> tensorproduct(A, x)
[[x, 2*x], [3*x, 4*x]]
>>> tensorproduct(A, B, B)
[[[[x**2, x*y], [x*y, y**2]], [[2*x**2, 2*x*y], [2*x*y, 2*y**2]]],␣
↪[[[3*x**2, 3*x*y], [3*x*y, 3*y**2]], [[4*x**2, 4*x*y], [4*x*y,␣
↪4*y**2]]]]
```

Applying this function on two matrices will result in a rank 4 array.

```
>>> from sympy import Matrix, eye
>>> m = Matrix([[x, y], [z, t]])
>>> p = tensorproduct(eye(3), m)
>>> p
[[[[x, y], [z, t]], [[0, 0], [0, 0]], [[0, 0], [0, 0]]], [[[0, 0], [0,␣
↪0]], [[x, y], [z, t]], [[0, 0], [0, 0]]], [[[0, 0], [0, 0]], [[0, 0],␣
↪[0, 0]], [[x, y], [z, t]]]]
```

sympy.tensor.array.**tensordiagonal**(*array*, **diagonal_axes*)

Diagonalization of an array-like object on the specified axes.

This is equivalent to multiplying the expression by Kronecker deltas uniting the axes.

The diagonal indices are put at the end of the axes.

Examples

`tensordiagonal` acting on a 2-dimensional array by axes 0 and 1 is equivalent to the diagonal of the matrix:

```
>>> from sympy import Array, tensordiagonal
>>> from sympy import Matrix, eye
>>> tensordiagonal(eye(3), (0, 1))
[1, 1, 1]
```

```
>>> from sympy.abc import a,b,c,d
>>> m1 = Matrix([[a, b], [c, d]])
>>> tensordiagonal(m1, [0, 1])
[a, d]
```

In case of higher dimensional arrays, the diagonalized out dimensions are appended removed and appended as a single dimension at the end:

```
>>> A = Array(range(18), (3, 2, 3))
>>> A
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]], [[12, 13, 14], [15,␣
↪16, 17]]]
>>> tensordiagonal(A, (0, 2))
[[0, 7, 14], [3, 10, 17]]
>>> from sympy import permutedims
>>> tensordiagonal(A, (0, 2)) == permutedims(Array([A[0, :, 0], A[1, :,␣
↪1], A[2, :, 2]]), [1, 0])
True
```


N-dim array expressions

Array expressions are expressions representing N-dimensional arrays, without evaluating them. These expressions represent in a certain way abstract syntax trees of operations on N-dimensional arrays.

Every N-dimensional array operator has a corresponding array expression object.

Table of correspondences:

Array operator	Array expression operator
tensorproduct	ArrayTensorProduct
tensorcontraction	ArrayContraction
tensordiagonal	ArrayDiagonal
permutedims	PermuteDims

Examples

`ArraySymbol` objects are the N-dimensional equivalent of `MatrixSymbol` objects in the matrix module:

```
>>> from sympy.tensor.array.expressions import ArraySymbol
>>> from sympy.abc import i, j, k
>>> A = ArraySymbol("A", (3, 2, 4))
>>> A.shape
(3, 2, 4)
>>> A[i, j, k]
A[i, j, k]
>>> A.as_explicit()
[[[A[0, 0, 0], A[0, 0, 1], A[0, 0, 2], A[0, 0, 3]],
  [A[0, 1, 0], A[0, 1, 1], A[0, 1, 2], A[0, 1, 3]]],
 [[A[1, 0, 0], A[1, 0, 1], A[1, 0, 2], A[1, 0, 3]],
  [A[1, 1, 0], A[1, 1, 1], A[1, 1, 2], A[1, 1, 3]]],
 [[A[2, 0, 0], A[2, 0, 1], A[2, 0, 2], A[2, 0, 3]],
  [A[2, 1, 0], A[2, 1, 1], A[2, 1, 2], A[2, 1, 3]]]]
```

Component-explicit arrays can be added inside array expressions:

```
>>> from sympy import Array
>>> from sympy import tensorproduct
>>> from sympy.tensor.array.expressions import ArrayTensorProduct
>>> a = Array([1, 2, 3])
>>> b = Array([i, j, k])
>>> expr = ArrayTensorProduct(a, b, b)
>>> expr
ArrayTensorProduct([1, 2, 3], [i, j, k], [i, j, k])
>>> expr.as_explicit() == tensorproduct(a, b, b)
True
```

Constructing array expressions from index-explicit forms

Array expressions are index-implicit. This means they do not use any indices to represent array operations. The function `convert_indexed_to_array( ... )` may be used to convert index-explicit expressions to array expressions. It takes as input two parameters: the index-explicit expression and the order of the indices:

```
>>> from sympy.tensor.array.expressions import convert_indexed_to_array
>>> from sympy import Sum
>>> A = ArraySymbol("A", (3, 3))
>>> B = ArraySymbol("B", (3, 3))
>>> convert_indexed_to_array(A[i, j], [i, j])
A
>>> convert_indexed_to_array(A[i, j], [j, i])
PermuteDims(A, (0 1))
>>> convert_indexed_to_array(A[i, j] + B[j, i], [i, j])
ArrayAdd(A, PermuteDims(B, (0 1)))
>>> convert_indexed_to_array(Sum(A[i, j]*B[j, k], (j, 0, 2)), [i, k])
ArrayContraction(ArrayTensorProduct(A, B), (1, 2))
```