The diagonal of a matrix in the array expression form:

```
>>> convert_indexed_to_array(A[i, i], [i])
ArrayDiagonal(A, (0, 1))
```

The trace of a matrix in the array expression form:

```
>>> convert_indexed_to_array(Sum(A[i, i], (i, 0, 2)), [i])
ArrayContraction(A, (0, 1))
```

Compatibility with matrices

Array expressions can be mixed with objects from the matrix module:

```
>>> from sympy import MatrixSymbol
>>> from sympy.tensor.array.expressions import ArrayContraction
>>> M = MatrixSymbol("M", 3, 3)
>>> N = MatrixSymbol("N", 3, 3)
```

Express the matrix product in the array expression form:

```
>>> from sympy.tensor.array.expressions import convert_matrix_to_array
>>> expr = convert_matrix_to_array(M*N)
>>> expr
ArrayContraction(ArrayTensorProduct(M, N), (1, 2))
```

The expression can be converted back to matrix form:

```
>>> from sympy.tensor.array.expressions import convert_array_to_matrix
>>> convert_array_to_matrix(expr)
M*N
```

Add a second contraction on the remaining axes in order to get the trace of $M \cdot N$:

```
>>> expr_tr = ArrayContraction(expr, (0, 1))
>>> expr_tr
ArrayContraction(ArrayContraction(ArrayTensorProduct(M, N), (1, 2)), (0, 1))
```

Flatten the expression by calling .doit() and remove the nested array contraction operations:

```
>>> expr_tr.doit()
ArrayContraction(ArrayTensorProduct(M, N), (0, 3), (1, 2))
```

Get the explicit form of the array expression:

```
>>> expr.as_explicit()
[[M[0, 0]*N[0, 0] + M[0, 1]*N[1, 0] + M[0, 2]*N[2, 0], M[0, 0]*N[0, 1] + M[0,␣
↪1]*N[1, 1] + M[0, 2]*N[2, 1], M[0, 0]*N[0, 2] + M[0, 1]*N[1, 2] + M[0,␣
↪2]*N[2, 2]],
 [M[1, 0]*N[0, 0] + M[1, 1]*N[1, 0] + M[1, 2]*N[2, 0], M[1, 0]*N[0, 1] + M[1,␣
↪1]*N[1, 1] + M[1, 2]*N[2, 1], M[1, 0]*N[0, 2] + M[1, 1]*N[1, 2] + M[1,␣
↪2]*N[2, 2]],
```

(continues on next page)

(continued from previous page)

```
 [M[2, 0]*N[0, 0] + M[2, 1]*N[1, 0] + M[2, 2]*N[2, 0], M[2, 0]*N[0, 1] + M[2,␣
↪1]*N[1, 1] + M[2, 2]*N[2, 1], M[2, 0]*N[0, 2] + M[2, 1]*N[1, 2] + M[2,␣
↪2]*N[2, 2]]]
```

Express the trace of a matrix:

```
>>> from sympy import Trace
>>> convert_matrix_to_array(Trace(M))
ArrayContraction(M, (0, 1))
>>> convert_matrix_to_array(Trace(M*N))
ArrayContraction(ArrayTensorProduct(M, N), (0, 3), (1, 2))
```

Express the transposition of a matrix (will be expressed as a permutation of the axes:

```
>>> convert_matrix_to_array(M.T)
PermuteDims(M, (0 1))
```

Compute the derivative array expressions:

```
>>> from sympy.tensor.array.expressions import array_derive
>>> d = array_derive(M, M)
>>> d
PermuteDims(ArrayTensorProduct(I, I), (3)(1 2))
```

Verify that the derivative corresponds to the form computed with explicit matrices:

```
>>> d.as_explicit()
[[[[1, 0, 0], [0, 0, 0], [0, 0, 0]], [[0, 1, 0], [0, 0, 0], [0, 0, 0]], [[0,␣
↪0, 1], [0, 0, 0], [0, 0, 0]]], [[[0, 0, 0], [1, 0, 0], [0, 0, 0]], [[0, 0,␣
↪0], [0, 1, 0], [0, 0, 0]], [[0, 0, 0], [0, 0, 1], [0, 0, 0]]], [[[0, 0, 0],␣
↪[0, 0, 0], [1, 0, 0]], [[0, 0, 0], [0, 0, 0], [0, 1, 0]], [[0, 0, 0], [0, 0,
↪ 0], [0, 0, 1]]]]
>>> Me = M.as_explicit()
>>> Me.diff(Me)
[[[[1, 0, 0], [0, 0, 0], [0, 0, 0]], [[0, 1, 0], [0, 0, 0], [0, 0, 0]], [[0,␣
↪0, 1], [0, 0, 0], [0, 0, 0]]], [[[0, 0, 0], [1, 0, 0], [0, 0, 0]], [[0, 0,␣
↪0], [0, 1, 0], [0, 0, 0]], [[0, 0, 0], [0, 0, 1], [0, 0, 0]]], [[[0, 0, 0],␣
↪[0, 0, 0], [1, 0, 0]], [[0, 0, 0], [0, 0, 0], [0, 1, 0]], [[0, 0, 0], [0, 0,
↪ 0], [0, 0, 1]]]]
```

Indexed Objects

Module that defines indexed objects

The classes `IndexedBase`, `Indexed`, and `Idx` represent a matrix element `M[i, j]` as in the following diagram:

```
1) The Indexed class represents the entire indexed object.
           |
        ___|___
       '       '
        M[i, j]
```

(continues on next page)

(continued from previous page)

```
       /   \__\______
       |             |
       |             |
       |     2) The Idx class represents indices; each Idx can
       |        optionally contain information about its range.
       |
 3) IndexedBase represents the 'stem' of an indexed object, here `M`.
    The stem used by itself is usually taken to represent the entire
    array.
```

There can be any number of indices on an Indexed object. No transformation properties are implemented in these Base objects, but implicit contraction of repeated indices is supported.

Note that the support for complicated (i.e. non-atomic) integer expressions as indices is limited. (This should be improved in future releases.)

Examples

To express the above matrix element example you would write:

```
>>> from sympy import symbols, IndexedBase, Idx
>>> M = IndexedBase('M')
>>> i, j = symbols('i j', cls=Idx)
>>> M[i, j]
M[i, j]
```

Repeated indices in a product implies a summation, so to express a matrix-vector product in terms of Indexed objects:

```
>>> x = IndexedBase('x')
>>> M[i, j]*x[j]
M[i, j]*x[j]
```

If the indexed objects will be converted to component based arrays, e.g. with the code printers or the autowrap framework, you also need to provide (symbolic or numerical) dimensions. This can be done by passing an optional shape parameter to IndexedBase upon construction:

```
>>> dim1, dim2 = symbols('dim1 dim2', integer=True)
>>> A = IndexedBase('A', shape=(dim1, 2*dim1, dim2))
>>> A.shape
(dim1, 2*dim1, dim2)
>>> A[i, j, 3].shape
(dim1, 2*dim1, dim2)
```

If an IndexedBase object has no shape information, it is assumed that the array is as large as the ranges of its indices:

```
>>> n, m = symbols('n m', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> M[i, j].shape
(m, n)
```

(continues on next page)

(continued from previous page)

```
>>> M[i, j].ranges
[(0, m - 1), (0, n - 1)]
```

The above can be compared with the following:

```
>>> A[i, 2, j].shape
(dim1, 2*dim1, dim2)
>>> A[i, 2, j].ranges
[(0, m - 1), None, (0, n - 1)]
```

To analyze the structure of indexed expressions, you can use the methods get_indices() and get_contraction_structure():

```
>>> from sympy.tensor import get_indices, get_contraction_structure
>>> get_indices(A[i, j, j])
({i}, {})
>>> get_contraction_structure(A[i, j, j])
{(j,): {A[i, j, j]}}
```

See the appropriate docstrings for a detailed explanation of the output.

class sympy.tensor.indexed.**Idx**(*label*, *range=None*, ***kw_args*)

Represents an integer index as an `Integer` or integer expression.

There are a number of ways to create an `Idx` object. The constructor takes two arguments:

label
An integer or a symbol that labels the index.

range
Optionally you can specify a range as either

- `Symbol` or integer: This is interpreted as a dimension. Lower and upper bounds are set to `0` and `range - 1`, respectively.
- `tuple`: The two elements are interpreted as the lower and upper bounds of the range, respectively.

Note: bounds of the range are assumed to be either integer or infinite (oo and -oo are allowed to specify an unbounded range). If `n` is given as a bound, then `n.is_integer` must not return false.

For convenience, if the label is given as a string it is automatically converted to an integer symbol. (Note: this conversion is not done for range or dimension arguments.)

Examples

```
>>> from sympy import Idx, symbols, oo
>>> n, i, L, U = symbols('n i L U', integer=True)
```

If a string is given for the label an integer `Symbol` is created and the bounds are both `None`:

```
>>> idx = Idx('qwerty'); idx
qwerty
>>> idx.lower, idx.upper
(None, None)
```

Both upper and lower bounds can be specified:

```
>>> idx = Idx(i, (L, U)); idx
i
>>> idx.lower, idx.upper
(L, U)
```

When only a single bound is given it is interpreted as the dimension and the lower bound defaults to 0:

```
>>> idx = Idx(i, n); idx.lower, idx.upper
(0, n - 1)
>>> idx = Idx(i, 4); idx.lower, idx.upper
(0, 3)
>>> idx = Idx(i, oo); idx.lower, idx.upper
(0, oo)
```

property label

Returns the label (Integer or integer expression) of the Idx object.

Examples

```
>>> from sympy import Idx, Symbol
>>> x = Symbol('x', integer=True)
>>> Idx(x).label
x
>>> j = Symbol('j', integer=True)
>>> Idx(j).label
j
>>> Idx(j + 1).label
j + 1
```

property lower

Returns the lower bound of the `Idx`.

Examples

```
>>> from sympy import Idx
>>> Idx('j', 2).lower
0
>>> Idx('j', 5).lower
0
>>> Idx('j').lower is None
True
```

property upper

Returns the upper bound of the Idx.

Examples

```
>>> from sympy import Idx
>>> Idx('j', 2).upper
1
>>> Idx('j', 5).upper
4
>>> Idx('j').upper is None
True
```

class sympy.tensor.indexed.**Indexed**(*base*, **args*, ***kw_args*)

Represents a mathematical object with indices.

```
>>> from sympy import Indexed, IndexedBase, Idx, symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j)
A[i, j]
```

It is recommended that Indexed objects be created by indexing IndexedBase: IndexedBase('A')[i, j] instead of Indexed(IndexedBase('A'), i, j).

```
>>> A = IndexedBase('A')
>>> a_ij = A[i, j]           # Prefer this,
>>> b_ij = Indexed(A, i, j)  # over this.
>>> a_ij == b_ij
True
```

property base

Returns the IndexedBase of the Indexed object.

Examples

```
>>> from sympy import Indexed, IndexedBase, Idx, symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j).base
A
>>> B = IndexedBase('B')
>>> B == B[i, j].base
True
```

property indices

Returns the indices of the Indexed object.

Examples

```
>>> from sympy import Indexed, Idx, symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j).indices
(i, j)
```

property ranges

Returns a list of tuples with lower and upper range of each index.

If an index does not define the data members upper and lower, the corresponding slot in the list contains None instead of a tuple.

Examples

```
>>> from sympy import Indexed,Idx, symbols
>>> Indexed('A', Idx('i', 2), Idx('j', 4), Idx('k', 8)).ranges
[(0, 1), (0, 3), (0, 7)]
>>> Indexed('A', Idx('i', 3), Idx('j', 3), Idx('k', 3)).ranges
[(0, 2), (0, 2), (0, 2)]
>>> x, y, z = symbols('x y z', integer=True)
>>> Indexed('A', x, y, z).ranges
[None, None, None]
```

property rank

Returns the rank of the Indexed object.

Examples

```
>>> from sympy import Indexed, Idx, symbols
>>> i, j, k, l, m = symbols('i:m', cls=Idx)
>>> Indexed('A', i, j).rank
2
>>> q = Indexed('A', i, j, k, l, m)
>>> q.rank
5
>>> q.rank == len(q.indices)
True
```

property shape

Returns a list with dimensions of each index.

Dimensions is a property of the array, not of the indices. Still, if the IndexedBase does not define a shape attribute, it is assumed that the ranges of the indices correspond to the shape of the array.

```
>>> from sympy import IndexedBase, Idx, symbols
>>> n, m = symbols('n m', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', m)
```

(continues on next page)

(continued from previous page)

```
>>> A = IndexedBase('A', shape=(n, n))
>>> B = IndexedBase('B')
>>> A[i, j].shape
(n, n)
>>> B[i, j].shape
(m, m)
```

class sympy.tensor.indexed.**IndexedBase**(*label*, *shape=None*, ***, *offset=0*, *strides=None*, ***kw_args*)

Represent the base or stem of an indexed object

The IndexedBase class represent an array that contains elements. The main purpose of this class is to allow the convenient creation of objects of the Indexed class. The __getitem__ method of IndexedBase returns an instance of Indexed. Alone, without indices, the IndexedBase class can be used as a notation for e.g. matrix equations, resembling what you could do with the Symbol class. But, the IndexedBase class adds functionality that is not available for Symbol instances:

- An IndexedBase object can optionally store shape information. This can be used in to check array conformance and conditions for numpy broadcasting. (TODO)
- An IndexedBase object implements syntactic sugar that allows easy symbolic representation of array operations, using implicit summation of repeated indices.
- The IndexedBase object symbolizes a mathematical structure equivalent to arrays, and is recognized as such for code generation and automatic compilation and wrapping.

```
>>> from sympy.tensor import IndexedBase, Idx
>>> from sympy import symbols
>>> A = IndexedBase('A'); A
A
>>> type(A)
<class 'sympy.tensor.indexed.IndexedBase'>
```

When an IndexedBase object receives indices, it returns an array with named axes, represented by an Indexed object:

```
>>> i, j = symbols('i j', integer=True)
>>> A[i, j, 2]
A[i, j, 2]
>>> type(A[i, j, 2])
<class 'sympy.tensor.indexed.Indexed'>
```

The IndexedBase constructor takes an optional shape argument. If given, it overrides any shape information in the indices. (But not the index ranges!)

```
>>> m, n, o, p = symbols('m n o p', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> A[i, j].shape
(m, n)
>>> B = IndexedBase('B', shape=(o, p))
>>> B[i, j].shape
(o, p)
```

Assumptions can be specified with keyword arguments the same way as for Symbol:

```
>>> A_real = IndexedBase('A', real=True)
>>> A_real.is_real
True
>>> A != A_real
True
```

Assumptions can also be inherited if a Symbol is used to initialize the IndexedBase:

```
>>> I = symbols('I', integer=True)
>>> C_inherit = IndexedBase(I)
>>> C_explicit = IndexedBase('I', integer=True)
>>> C_inherit == C_explicit
True
```

property label

Returns the label of the IndexedBase object.

Examples

```
>>> from sympy import IndexedBase
>>> from sympy.abc import x, y
>>> IndexedBase('A', shape=(x, y)).label
A
```

property offset

Returns the offset for the IndexedBase object.

This is the value added to the resulting index when the 2D Indexed object is unrolled to a 1D form. Used in code generation.

Examples

```
>>> from sympy.printing import ccode
>>> from sympy.tensor import IndexedBase, Idx
>>> from sympy import symbols
>>> l, m, n, o = symbols('l m n o', integer=True)
>>> A = IndexedBase('A', strides=(l, m, n), offset=o)
>>> i, j, k = map(Idx, 'ijk')
>>> ccode(A[i, j, k])
'A[l*i + m*j + n*k + o]'
```

property shape

Returns the shape of the IndexedBase object.

Examples

```
>>> from sympy import IndexedBase, Idx
>>> from sympy.abc import x, y
>>> IndexedBase('A', shape=(x, y)).shape
(x, y)
```

Note: If the shape of the IndexedBase is specified, it will override any shape information given by the indices.

```
>>> A = IndexedBase('A', shape=(x, y))
>>> B = IndexedBase('B')
>>> i = Idx('i', 2)
>>> j = Idx('j', 1)
>>> A[i, j].shape
(x, y)
>>> B[i, j].shape
(2, 1)
```

property strides

Returns the strided scheme for the IndexedBase object.

Normally this is a tuple denoting the number of steps to take in the respective dimension when traversing an array. For code generation purposes strides='C' and strides='F' can also be used.

strides='C' would mean that code printer would unroll in row-major order and 'F' means unroll in column major order.

Methods

Module with functions operating on IndexedBase, Indexed and Idx objects

- Check shape conformance
- Determine indices in resulting expression

etc.

Methods in this module could be implemented by calling methods on Expr objects instead. When things stabilize this could be a useful refactoring.

sympy.tensor.index_methods.**get_contraction_structure**(*expr*)

Determine dummy indices of expr and describe its structure

By *dummy* we mean indices that are summation indices.

The structure of the expression is determined and described as follows:

1) A conforming summation of Indexed objects is described with a dict where the keys are summation indices and the corresponding values are sets containing all terms for which the summation applies. All Add objects in the SymPy expression tree are described like this.

2) For all nodes in the SymPy expression tree that are *not* of type Add, the following applies:

If a node discovers contractions in one of its arguments, the node itself will be stored as a key in the dict. For that key, the corresponding value is a list of dicts, each of which is the result of a recursive call to get_contraction_structure(). The list contains only dicts for the non-trivial deeper contractions, omitting dicts with None as the one and only key.

Note: The presence of expressions among the dictionary keys indicates multiple levels of index contractions. A nested dict displays nested contractions and may itself contain dicts from a deeper level. In practical calculations the summation in the deepest nested level must be calculated first so that the outer expression can access the resulting indexed object.

Examples

```
>>> from sympy.tensor.index_methods import get_contraction_structure
>>> from sympy import default_sort_key
>>> from sympy.tensor import IndexedBase, Idx
>>> x, y, A = map(IndexedBase, ['x', 'y', 'A'])
>>> i, j, k, l = map(Idx, ['i', 'j', 'k', 'l'])
>>> get_contraction_structure(x[i]*y[i] + A[j, j])
{(i,): {x[i]*y[i]}, (j,): {A[j, j]}}
>>> get_contraction_structure(x[i]*y[j])
{None: {x[i]*y[j]}}
```

A multiplication of contracted factors results in nested dicts representing the internal contractions.

```
>>> d = get_contraction_structure(x[i, i]*y[j, j])
>>> sorted(d.keys(), key=default_sort_key)
[None, x[i, i]*y[j, j]]
```

In this case, the product has no contractions:

```
>>> d[None]
{x[i, i]*y[j, j]}
```

Factors are contracted "first":

```
>>> sorted(d[x[i, i]*y[j, j]], key=default_sort_key)
[{(i,): {x[i, i]}}, {(j,): {y[j, j]}}]
```

A parenthesized Add object is also returned as a nested dictionary. The term containing the parenthesis is a Mul with a contraction among the arguments, so it will be found as a key in the result. It stores the dictionary resulting from a recursive call on the Add expression.

```
>>> d = get_contraction_structure(x[i]*(y[i] + A[i, j]*x[j]))
>>> sorted(d.keys(), key=default_sort_key)
[(A[i, j]*x[j] + y[i])*x[i], (i,)]
>>> d[(i,)]
{(A[i, j]*x[j] + y[i])*x[i]}
```

(continues on next page)

(continued from previous page)

```
>>> d[x[i]*(A[i, j]*x[j] + y[i])]
[{None: {y[i]}, (j,): {A[i, j]*x[j]}}]
```

Powers with contractions in either base or exponent will also be found as keys in the dictionary, mapping to a list of results from recursive calls:

```
>>> d = get_contraction_structure(A[j, j]**A[i, i])
>>> d[None]
{A[j, j]**A[i, i]}
>>> nested_contractions = d[A[j, j]**A[i, i]]
>>> nested_contractions[0]
{(j,): {A[j, j]}}
>>> nested_contractions[1]
{(i,): {A[i, i]}}
```

The description of the contraction structure may appear complicated when represented with a string in the above examples, but it is easy to iterate over:

```
>>> from sympy import Expr
>>> for key in d:
...     if isinstance(key, Expr):
...         continue
...     for term in d[key]:
...         if term in d:
...             # treat deepest contraction first
...             pass
...     # treat outermost contactions here
```

sympy.tensor.index_methods.**get_indices**(*expr*)

Determine the outer indices of expression `expr`

By *outer* we mean indices that are not summation indices. Returns a set and a dict. The set contains outer indices and the dict contains information about index symmetries.

Examples

```
>>> from sympy.tensor.index_methods import get_indices
>>> from sympy import symbols
>>> from sympy.tensor import IndexedBase
>>> x, y, A = map(IndexedBase, ['x', 'y', 'A'])
>>> i, j, a, z = symbols('i j a z', integer=True)
```

The indices of the total expression is determined, Repeated indices imply a summation, for instance the trace of a matrix A:

```
>>> get_indices(A[i, i])
(set(), {})
```

In the case of many terms, the terms are required to have identical outer indices. Else an IndexConformanceException is raised.

```
>>> get_indices(x[i] + A[i, j]*y[j])
({i}, {})
```

Exceptions

An IndexConformanceException means that the terms ar not compatible, e.g.

```
>>> get_indices(x[i] + y[j])
        (...)
IndexConformanceException: Indices are not consistent: x(i) + y(j)
```

Warning: The concept of *outer* indices applies recursively, starting on the deepest level. This implies that dummies inside parenthesis are assumed to be summed first, so that the following expression is handled gracefully:

```
>>> get_indices((x[i] + A[i, j]*y[j])*x[j])
({i, j}, {})
```

This is correct and may appear convenient, but you need to be careful with this as SymPy will happily .expand() the product, if requested. The resulting expression would mix the outer j with the dummies inside the parenthesis, which makes it a different expression. To be on the safe side, it is best to avoid such ambiguities by using unique indices for all contractions that should be held separate.

Tensor

class sympy.tensor.tensor.**TensorIndexType**(*name, dummy_name=None, dim=None, eps_dim=None, metric_symmetry=1, metric_name='metric', **kwargs*)

A TensorIndexType is characterized by its name and its metric.

Parameters

name : name of the tensor type

dummy_name : name of the head of dummy indices

dim : dimension, it can be a symbol or an integer or None

eps_dim : dimension of the epsilon tensor

metric_symmetry : integer that denotes metric symmetry or None for no metric

metric_name : string with the name of the metric tensor

Notes

The possible values of the `metric_symmetry` parameter are:

> `1` : metric tensor is fully symmetric `0` : metric tensor possesses no index symmetry `-1` : metric tensor is fully antisymmetric `None`: there is no metric tensor (metric equals to `None`)

The metric is assumed to be symmetric by default. It can also be set to a custom tensor by the `.set_metric()` method.

If there is a metric the metric is used to raise and lower indices.

In the case of non-symmetric metric, the following raising and lowering conventions will be adopted:

`psi(a) = g(a, b)*psi(-b); chi(-a) = chi(b)*g(-b, -a)`

From these it is easy to find:

`g(-a, b) = delta(-a, b)`

where `delta(-a, b) = delta(b, -a)` is the `Kronecker delta` (see `TensorIndex` for the conventions on indices). For antisymmetric metrics there is also the following equality:

`g(a, -b) = -delta(a, -b)`

If there is no metric it is not possible to raise or lower indices; e.g. the index of the defining representation of `SU(N)` is 'covariant' and the conjugate representation is 'contravariant'; for `N > 2` they are linearly independent.

`eps_dim` is by default equal to `dim`, if the latter is an integer; else it can be assigned (for use in naive dimensional regularization); if `eps_dim` is not an integer `epsilon` is `None`.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> Lorentz.metric
metric(Lorentz,Lorentz)
```

Attributes

`metric`	(the metric tensor)
`delta`	(`Kronecker delta`)
`epsilon`	(the `Levi-Civita epsilon` tensor)
`data`	((deprecated) a property to add `ndarray` values, to work in a specified basis.)

class `sympy.tensor.tensor.`**`TensorIndex`**(*name*, *tensor_index_type*, *is_up=True*)

Represents a tensor index

Parameters

name : name of the index, or `True` if you want it to be automatically assigned

tensor_index_type : `TensorIndexType` of the index

is_up : flag for contravariant index (is_up=True by default)

Notes

Tensor indices are contracted with the Einstein summation convention.

An index can be in contravariant or in covariant form; in the latter case it is represented prepending a `-` to the index name. Adding `-` to a covariant (is_up=False) index makes it contravariant.

Dummy indices have a name with head given by `tensor_inde_type.dummy_name` with underscore and a number.

Similar to `symbols` multiple contravariant indices can be created at once using `tensor_indices(s, typ)`, where `s` is a string of names.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, TensorIndex,␣
↪TensorHead, tensor_indices
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> mu = TensorIndex('mu', Lorentz, is_up=False)
>>> nu, rho = tensor_indices('nu, rho', Lorentz)
>>> A = TensorHead('A', [Lorentz, Lorentz])
>>> A(mu, nu)
A(-mu, nu)
>>> A(-mu, -rho)
A(mu, -rho)
>>> A(mu, -mu)
A(-L_0, L_0)
```

Attributes

name	
tensor_index_type	
is_up	

class `sympy.tensor.tensor.`**`TensorHead`**(*name, index_types, symmetry=None, comm=0*)

Tensor head of the tensor.

Parameters

name : name of the tensor

index_types : list of TensorIndexType

symmetry : TensorSymmetry of the tensor

comm : commutation group number

Notes

Similar to symbols multiple TensorHeads can be created using `tensorhead(s, typ, sym=None, comm=0)` function, where `s` is the string of names and `sym` is the monoterm tensor symmetry (see `tensorsymmetry`).

A `TensorHead` belongs to a commutation group, defined by a symbol on number `comm` (see `_TensorManager.set_comm`); tensors in a commutation group have the same commutation properties; by default `comm` is `0`, the group of the commuting tensors.

Examples

Define a fully antisymmetric tensor of rank 2:

```
>>> from sympy.tensor.tensor import TensorIndexType, TensorHead,␣
↪TensorSymmetry
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> asym2 = TensorSymmetry.fully_symmetric(-2)
>>> A = TensorHead('A', [Lorentz, Lorentz], asym2)
```

Examples with ndarray values, the components data assigned to the `TensorHead` object are assumed to be in a fully-contravariant representation. In case it is necessary to assign components data which represents the values of a non-fully covariant tensor, see the other examples.

```
>>> from sympy.tensor.tensor import tensor_indices
>>> from sympy import diag
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> i0, i1 = tensor_indices('i0:2', Lorentz)
```

Specify a replacement dictionary to keep track of the arrays to use for replacements in the tensorial expression. The `TensorIndexType` is associated to the metric used for contractions (in fully covariant form):

```
>>> repl = {Lorentz: diag(1, -1, -1, -1)}
```

Let’s see some examples of working with components with the electromagnetic tensor:

```
>>> from sympy import symbols
>>> Ex, Ey, Ez, Bx, By, Bz = symbols('E_x E_y E_z B_x B_y B_z')
>>> c = symbols('c', positive=True)
```

Let’s define F, an antisymmetric tensor:

```
>>> F = TensorHead('F', [Lorentz, Lorentz], asym2)
```

Let’s update the dictionary to contain the matrix to use in the replacements:

```
>>> repl.update({F(-i0, -i1): [
... [0, Ex/c, Ey/c, Ez/c],
... [-Ex/c, 0, -Bz, By],
... [-Ey/c, Bz, 0, -Bx],
... [-Ez/c, -By, Bx, 0]]})
```

Now it is possible to retrieve the contravariant form of the Electromagnetic tensor:

```
>>> F(i0, i1).replace_with_arrays(repl, [i0, i1])
[[0, -E_x/c, -E_y/c, -E_z/c], [E_x/c, 0, -B_z, B_y], [E_y/c, B_z, 0, -B_
↪x], [E_z/c, -B_y, B_x, 0]]
```

and the mixed contravariant-covariant form:

```
>>> F(i0, -i1).replace_with_arrays(repl, [i0, -i1])
[[0, E_x/c, E_y/c, E_z/c], [E_x/c, 0, B_z, -B_y], [E_y/c, -B_z, 0, B_x],␣
↪[E_z/c, B_y, -B_x, 0]]
```

Energy-momentum of a particle may be represented as:

```
>>> from sympy import symbols
>>> P = TensorHead('P', [Lorentz], TensorSymmetry.no_symmetry(1))
>>> E, px, py, pz = symbols('E p_x p_y p_z', positive=True)
>>> repl.update({P(i0): [E, px, py, pz]})
```

The contravariant and covariant components are, respectively:

```
>>> P(i0).replace_with_arrays(repl, [i0])
[E, p_x, p_y, p_z]
>>> P(-i0).replace_with_arrays(repl, [-i0])
[E, -p_x, -p_y, -p_z]
```

The contraction of a 1-index tensor by itself:

```
>>> expr = P(i0)*P(-i0)
>>> expr.replace_with_arrays(repl, [])
E**2 - p_x**2 - p_y**2 - p_z**2
```

Attributes

name	
index_types	
rank	(total number of indices)
symmetry	
comm	(commutation group)

commutes_with(*other*)

Returns 0 if self and other commute, 1 if they anticommute.

Returns None if self and other neither commute nor anticommute.

sympy.tensor.tensor.**tensor_heads**(*s, index_types, symmetry=None, comm=0*)

Returns a sequence of TensorHeads from a string s

class sympy.tensor.tensor.**TensExpr**(**args*)

Abstract base class for tensor expressions

Notes

A tensor expression is an expression formed by tensors; currently the sums of tensors are distributed.

A `TensExpr` can be a `TensAdd` or a `TensMul`.

`TensMul` objects are formed by products of component tensors, and include a coefficient, which is a SymPy expression.

In the internal representation contracted indices are represented by `(ipos1, ipos2, icomp1, icomp2)`, where `icomp1` is the position of the component tensor with contravariant index, `ipos1` is the slot which the index occupies in that component tensor.

Contracted indices are therefore nameless in the internal representation.

get_matrix()

DEPRECATED: do not use.

Returns ndarray components data as a matrix, if components data are available and ndarray dimension does not exceed 2.

replace_with_arrays(*replacement_dict*, *indices=None*)

Replace the tensorial expressions with arrays. The final array will correspond to the N-dimensional array with indices arranged according to `indices`.

Parameters

replacement_dict

dictionary containing the replacement rules for tensors.

indices

the index order with respect to which the array is read. The original index order will be used if no value is passed.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices
>>> from sympy.tensor.tensor import TensorHead
>>> from sympy import symbols, diag
```

```
>>> L = TensorIndexType("L")
>>> i, j = tensor_indices("i j", L)
>>> A = TensorHead("A", [L])
>>> A(i).replace_with_arrays({A(i): [1, 2]}, [i])
[1, 2]
```

Since 'indices' is optional, we can also call replace_with_arrays by this way if no specific index order is needed:

```
>>> A(i).replace_with_arrays({A(i): [1, 2]})
[1, 2]
```

```
>>> expr = A(i)*A(j)
>>> expr.replace_with_arrays({A(i): [1, 2]})
[[1, 2], [2, 4]]
```

For contractions, specify the metric of the TensorIndexType, which in this case is L, in its covariant form:

```
>>> expr = A(i)*A(-i)
>>> expr.replace_with_arrays({A(i): [1, 2], L: diag(1, -1)})
-3
```

Symmetrization of an array:

```
>>> H = TensorHead("H", [L, L])
>>> a, b, c, d = symbols("a b c d")
>>> expr = H(i, j)/2 + H(j, i)/2
>>> expr.replace_with_arrays({H(i, j): [[a, b], [c, d]]})
[[a, b/2 + c/2], [b/2 + c/2, d]]
```

Anti-symmetrization of an array:

```
>>> expr = H(i, j)/2 - H(j, i)/2
>>> repl = {H(i, j): [[a, b], [c, d]]}
>>> expr.replace_with_arrays(repl)
[[0, b/2 - c/2], [-b/2 + c/2, 0]]
```

The same expression can be read as the transpose by inverting i and j:

```
>>> expr.replace_with_arrays(repl, [j, i])
[[0, -b/2 + c/2], [b/2 - c/2, 0]]
```

class sympy.tensor.tensor.**TensAdd**(**args*, ***kw_args*)

Sum of tensors.

Parameters

free_args : list of the free indices

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_heads,␣
↪tensor_indices
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> a, b = tensor_indices('a,b', Lorentz)
>>> p, q = tensor_heads('p,q', [Lorentz])
>>> t = p(a) + q(a); t
p(a) + q(a)
```

Examples with components data added to the tensor expression:

```
>>> from sympy import symbols, diag
>>> x, y, z, t = symbols("x y z t")
>>> repl = {}
>>> repl[Lorentz] = diag(1, -1, -1, -1)
>>> repl[p(a)] = [1, 2, 3, 4]
>>> repl[q(a)] = [x, y, z, t]
```

The following are: 2**2 - 3**2 - 2**2 - 7**2 ==> -58

```
>>> expr = p(a) + q(a)
>>> expr.replace_with_arrays(repl, [a])
[x + 1, y + 2, z + 3, t + 4]
```

Attributes

args	(tuple of addends)
rank	(rank of the tensor)
free_args	(list of the free indices in sorted order)

canon_bp()

Canonicalize using the Butler-Portugal algorithm for canonicalization under monoterm symmetries.

contract_metric(*g*)

Raise or lower indices with the metric g.

Parameters

g : metric

contract_all : if True, eliminate all g which are contracted

Notes

see the TensorIndexType docstring for the contraction conventions

class sympy.tensor.tensor.**TensMul**(**args*, ***kw_args*)

Product of tensors.

Parameters

coeff : SymPy coefficient of the tensor

args

Notes

args[0] list of TensorHead of the component tensors.

args[1] list of (ind, ipos, icomp) where ind is a free index, ipos is the slot position of ind in the icomp-th component tensor.

args[2] list of tuples representing dummy indices. (ipos1, ipos2, icomp1, icomp2) indicates that the contravariant dummy index is the ipos1-th slot position in the icomp1-th component tensor; the corresponding covariant index is in the ipos2 slot position in the icomp2-th component tensor.

Attributes

components	(list of TensorHead of the component tensors)
types	(list of nonrepeated TensorIndexType)
free	(list of (ind, ipos, icomp), see Notes)
dum	(list of (ipos1, ipos2, icomp1, icomp2), see Notes)
ext_rank	(rank of the tensor counting the dummy indices)
rank	(rank of the tensor)
coeff	(SymPy coefficient of the tensor)
free_args	(list of the free indices in sorted order)
is_canon_bp	(True if the tensor in in canonical form)

canon_bp()

Canonicalize using the Butler-Portugal algorithm for canonicalization under monoterm symmetries.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices,␣
↪TensorHead, TensorSymmetry
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> A = TensorHead('A', [Lorentz]*2, TensorSymmetry.fully_symmetric(-
↪2))
>>> t = A(m0,-m1)*A(m1,-m0)
>>> t.canon_bp()
-A(L_0, L_1)*A(-L_0, -L_1)
>>> t = A(m0,-m1)*A(m1,-m2)*A(m2,-m0)
>>> t.canon_bp()
0
```

contract_metric(g)

Raise or lower indices with the metric g.

Parameters

g : metric

Notes

See the TensorIndexType docstring for the contraction conventions.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices,␣
↪tensor_heads
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensor_heads('p,q', [Lorentz])
>>> t = p(m0)*q(m1)*g(-m0, -m1)
>>> t.canon_bp()
metric(L_0, L_1)*p(-L_0)*q(-L_1)
>>> t.contract_metric(g).canon_bp()
p(L_0)*q(-L_0)
```

get_free_indices() → List[*TensorIndex* (page 1410)]

Returns the list of free indices of the tensor.

Explanation

The indices are listed in the order in which they appear in the component tensors.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices,␣
↪tensor_heads
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensor_heads('p,q', [Lorentz])
>>> t = p(m1)*g(m0,m2)
>>> t.get_free_indices()
[m1, m0, m2]
>>> t2 = p(m1)*g(-m1, m2)
>>> t2.get_free_indices()
[m2]
```

get_indices()

Returns the list of indices of the tensor.

Explanation

The indices are listed in the order in which they appear in the component tensors. The dummy indices are given a name which does not collide with the names of the free indices.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices,␣
↪tensor_heads
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensor_heads('p,q', [Lorentz])
>>> t = p(m1)*g(m0,m2)
>>> t.get_indices()
[m1, m0, m2]
>>> t2 = p(m1)*g(-m1, m2)
>>> t2.get_indices()
[L_0, -L_0, m2]
```

perm2tensor(*g*, *is_canon_bp=False*)

Returns the tensor corresponding to the permutation g

For further details, see the method in TIDS with the same name.

sorted_components()

Returns a tensor product with sorted components.

split()

Returns a list of tensors, whose product is self.

Explanation

Dummy indices contracted among different tensor components become free indices with the same name as the one used to represent the dummy indices.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices,␣
↪tensor_heads, TensorSymmetry
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> a, b, c, d = tensor_indices('a,b,c,d', Lorentz)
>>> A, B = tensor_heads('A,B', [Lorentz]*2, TensorSymmetry.fully_
↪symmetric(2))
>>> t = A(a,b)*B(-b,c)
>>> t
A(a, L_0)*B(-L_0, c)
>>> t.split()
[A(a, L_0), B(-L_0, c)]
```

sympy.tensor.tensor.**canon_bp**(*p*)

Butler-Portugal canonicalization. See tensor_can.py from the combinatorics module for the details.

sympy.tensor.tensor.**riemann_cyclic_replace**(*t_r*)

replace Riemann tensor with an equivalent expression

R(m,n,p,q) -> 2/3*R(m,n,p,q) - 1/3*R(m,q,n,p) + 1/3*R(m,p,n,q) `sympy.tensor.tensor.`**`riemann_cyclic`**`(`*t2*`)`

Replace each Riemann tensor with an equivalent expression satisfying the cyclic identity.

This trick is discussed in the reference guide to Cadabra.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices,␣
↪TensorHead, riemann_cyclic, TensorSymmetry
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> i, j, k, l = tensor_indices('i,j,k,l', Lorentz)
>>> R = TensorHead('R', [Lorentz]*4, TensorSymmetry.riemann())
>>> t = R(i,j,k,l)*(R(-i,-j,-k,-l) - 2*R(-i,-k,-j,-l))
>>> riemann_cyclic(t)
0
```

`class` `sympy.tensor.tensor.`**`TensorSymmetry`**`(`**args, **kw_args*`)`

Monoterm symmetry of a tensor (i.e. any symmetric or anti-symmetric index permutation). For the relevant terminology see `tensor_can.py` section of the combinatorics module.

Parameters

bsgs : tuple `(base, sgs)` BSGS of the symmetry of the tensor

Notes

A tensor can have an arbitrary monoterm symmetry provided by its BSGS. Multiterm symmetries, like the cyclic symmetry of the Riemann tensor (i.e., Bianchi identity), are not covered. See combinatorics module for information on how to generate BSGS for a general index permutation group. Simple symmetries can be generated using built-in methods.

Examples

Define a symmetric tensor of rank 2

```
>>> from sympy.tensor.tensor import TensorIndexType, TensorSymmetry, get_
↪symmetric_group_sgs, TensorHead
>>> Lorentz = TensorIndexType('Lorentz', dummy_name='L')
>>> sym = TensorSymmetry(get_symmetric_group_sgs(2))
>>> T = TensorHead('T', [Lorentz]*2, sym)
```

Note, that the same can also be done using built-in TensorSymmetry methods

```
>>> sym2 = TensorSymmetry.fully_symmetric(2)
>>> sym == sym2
True
```

See also:

`sympy.combinatorics.tensor_can.get_symmetric_group_sgs` (page 370)

Attributes

`base`	(base of the BSGS)
`generators`	(generators of the BSGS)
`rank`	(rank of the tensor)

classmethod direct_product(**args*)

Returns a TensorSymmetry object that is being a direct product of fully (anti-)symmetric index permutation groups.

Notes

Some examples for different values of `(*args)`: `(1)` vector, equivalent to `TensorSymmetry.fully_symmetric(1)` `(2)` tensor with 2 symmetric indices, equivalent to `.fully_symmetric(2)` `(-2)` tensor with 2 antisymmetric indices, equivalent to `.fully_symmetric(-2)` `(2, -2)` tensor with the first 2 indices commuting and the last 2 anticommuting `(1, 1, 1)` tensor with 3 indices without any symmetry

classmethod fully_symmetric(*rank*)

Returns a fully symmetric (antisymmetric if `rank``<0) TensorSymmetry object for ``abs(rank)` indices.

classmethod no_symmetry(*rank*)

TensorSymmetry object for `rank` indices with no symmetry

classmethod riemann()

Returns a monotorem symmetry of the Riemann tensor

`sympy.tensor.tensor.`**tensorsymmetry**(**args*)

Returns a `TensorSymmetry` object. This method is deprecated, use `TensorSymmetry.direct_product()` or `.riemann()` instead.

Explanation

One can represent a tensor with any monoterm slot symmetry group using a BSGS.

`args` can be a BSGS `args[0]` base `args[1]` sgs

Usually tensors are in (direct products of) representations of the symmetric group; `args` can be a list of lists representing the shapes of Young tableaux

Notes

For instance: `[[1]]` vector `[[1]*n]` symmetric tensor of rank `n` `[[n]]` antisymmetric tensor of rank `n` `[[2, 2]]` monoterm slot symmetry of the Riemann tensor `[[1],[1]]` vector*vector `[[2],[1],[1]` (antisymmetric tensor)*vector*vector

Notice that with the shape `[2, 2]` we associate only the monoterm symmetries of the Riemann tensor; this is an abuse of notation, since the shape `[2, 2]` corresponds usually to the irreducible representation characterized by the monoterm symmetries and by the cyclic symmetry.

class sympy.tensor.tensor.**TensorType**(**args*, ***kwargs*)

Class of tensor types. Deprecated, use tensor_heads() instead.

Parameters

index_types : list of `TensorIndexType` of the tensor indices

symmetry : `TensorSymmetry` of the tensor

Attributes

`index_types`	
`symmetry`	
`types`	(list of `TensorIndexType` without repetitions)

class sympy.tensor.tensor.**_TensorManager**

Class to manage tensor properties.

Notes

Tensors belong to tensor commutation groups; each group has a label `comm`; there are predefined labels:

`0` tensors commuting with any other tensor

`1` tensors anticommuting among themselves

`2` tensors not commuting, apart with those with `comm=0`

Other groups can be defined using `set_comm`; tensors in those groups commute with those with `comm=0`; by default they do not commute with any other group.

clear()

Clear the TensorManager.

comm_i2symbol(*i*)

Returns the symbol corresponding to the commutation group number.

comm_symbols2i(*i*)

Get the commutation group number corresponding to `i`.

`i` can be a symbol or a number or a string.

If `i` is not already defined its commutation group number is set.

get_comm(*i*, *j*)

Return the commutation parameter for commutation group numbers `i, j`

see `_TensorManager.set_comm`

set_comm(*i*, *j*, *c*)

Set the commutation parameter `c` for commutation groups `i, j`.

Parameters

i, j : symbols representing commutation groups

c : group commutation number

Notes

i, j can be symbols, strings or numbers, apart from 0, 1 and 2 which are reserved respectively for commuting, anticommuting tensors and tensors not commuting with any other group apart with the commuting tensors. For the remaining cases, use this method to set the commutation rules; by default c=None.

The group commutation number c is assigned in correspondence to the group commutation symbols; it can be

0 commuting

1 anticommuting

None no commutation property

Examples

G and GH do not commute with themselves and commute with each other; A is commuting.

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices,␣
↪TensorHead, TensorManager, TensorSymmetry
>>> Lorentz = TensorIndexType('Lorentz')
>>> i0,i1,i2,i3,i4 = tensor_indices('i0:5', Lorentz)
>>> A = TensorHead('A', [Lorentz])
>>> G = TensorHead('G', [Lorentz], TensorSymmetry.no_symmetry(1),
↪'Gcomm')
>>> GH = TensorHead('GH', [Lorentz], TensorSymmetry.no_symmetry(1),
↪'GHcomm')
>>> TensorManager.set_comm('Gcomm', 'GHcomm', 0)
>>> (GH(i1)*G(i0)).canon_bp()
G(i0)*GH(i1)
>>> (G(i1)*G(i0)).canon_bp()
G(i1)*G(i0)
>>> (G(i1)*A(i0)).canon_bp()
A(i0)*G(i1)
```

set_comms(**args*)

Set the commutation group numbers c for symbols i, j.

Parameters

args : sequence of (i, j, c)

Tensor Operators

class sympy.tensor.toperators.**PartialDerivative**(*expr*, **variables*)

Partial derivative for tensor expressions.

Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, TensorHead
>>> from sympy.tensor.toperators import PartialDerivative
>>> from sympy import symbols
>>> L = TensorIndexType("L")
>>> A = TensorHead("A", [L])
>>> B = TensorHead("B", [L])
>>> i, j, k = symbols("i j k")
```

```
>>> expr = PartialDerivative(A(i), A(j))
>>> expr
PartialDerivative(A(i), A(j))
```

The PartialDerivative object behaves like a tensorial expression:

```
>>> expr.get_indices()
[i, -j]
```

Notice that the deriving variables have opposite valence than the printed one: A(j) is printed as covariant, but the index of the derivative is actually contravariant, i.e. -j.

Indices can be contracted:

```
>>> expr = PartialDerivative(A(i), A(i))
>>> expr
PartialDerivative(A(L_0), A(L_0))
>>> expr.get_indices()
[L_0, -L_0]
```

The method .get_indices() always returns all indices (even the contracted ones). If only uncontracted indices are needed, call .get_free_indices():

```
>>> expr.get_free_indices()
[]
```

Nested partial derivatives are flattened:

```
>>> expr = PartialDerivative(PartialDerivative(A(i), A(j)), A(k))
>>> expr
PartialDerivative(A(i), A(j), A(k))
>>> expr.get_indices()
[i, -j, -k]
```

Replace a derivative with array values:

```
>>> from sympy.abc import x, y
>>> from sympy import sin, log
>>> compA = [sin(x), log(x)*y**3]
>>> compB = [x, y]
>>> expr = PartialDerivative(A(i), B(j))
>>> expr.replace_with_arrays({A(i): compA, B(i): compB})
[[cos(x), 0], [y**3/x, 3*y**2*log(x)]]
```

The returned array is indexed by $(i, -j)$.

Be careful that other SymPy modules put the indices of the deriving variables before the indices of the derivand in the derivative result. For example:

```
>>> expr.get_free_indices()
[i, -j]
```

```
>>> from sympy import Matrix, Array
>>> Matrix(compA).diff(Matrix(compB)).reshape(2, 2)
[[cos(x), y**3/x], [0, 3*y**2*log(x)]]
>>> Array(compA).diff(Array(compB))
[[cos(x), y**3/x], [0, 3*y**2*log(x)]]
```

These are the transpose of the result of `PartialDerivative`, as the matrix and the array modules put the index $-j$ before i in the derivative result. An array read with index order $(-j, i)$ is indeed the transpose of the same array read with index order $(i, -j)$. By specifying the index order to `.replace_with_arrays` one can get a compatible expression:

```
>>> expr.replace_with_arrays({A(i): compA, B(i): compB}, [-j, i])
[[cos(x), y**3/x], [0, 3*y**2*log(x)]]
```

Vector

The vector module provides tools for basic vector math and differential calculus with respect to 3D Cartesian coordinate systems. This documentation provides an overview of all the features offered, and relevant API.

Guide to Vector

Introduction

This page gives a brief conceptual overview of the functionality present in *sympy.vector* (page 1425).

Vectors and Scalars

In vector math, we deal with two kinds of quantities – scalars and vectors.

A **scalar** is an entity which only has a magnitude – no direction. Examples of scalar quantities include mass, electric charge, temperature, distance, etc.

A **vector**, on the other hand, is an entity that is characterized by a magnitude and a direction. Examples of vector quantities are displacement, velocity, magnetic field, etc.

A scalar can be depicted just by a number, for e.g. a temperature of 300 K. On the other hand, vectorial quantities like acceleration are usually denoted by a vector. Given a vector $\mathbf{V}$, the magnitude of the corresponding quantity can be calculated as the magnitude of the vector itself $\|\mathbf{V}\|$, while the direction would be specified by a unit vector in the direction of the original vector, $\mathbf{\hat{V}} = \frac{\mathbf{V}}{\|\mathbf{V}\|}$.

For example, consider a displacement of $(3\mathbf{\hat{i}} + 4\mathbf{\hat{j}} + 5\mathbf{\hat{k}})$ m, where , as per standard convention, $\mathbf{\hat{i}}$, $\mathbf{\hat{j}}$ and $\mathbf{\hat{k}}$ represent unit vectors along the $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$ axes respectively. Therefore, it can be concluded that the distance traveled is $\|3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}}\|$ m $= 5\sqrt{2}$ m. The direction of travel is given by the unit vector $\frac{3}{5\sqrt{2}}\hat{\mathbf{i}} + \frac{4}{5\sqrt{2}}\hat{\mathbf{j}} + \frac{5}{5\sqrt{2}}\hat{\mathbf{k}}$.

Coordinate Systems

A **coordinate system** is an abstract mathematical entity used to define the notion of directions and locations in n-dimensional spaces. This module deals with 3-dimensional spaces, with the conventional X, Y and Z axes defined with respect to each coordinate system.

Each coordinate system also has a special reference point called the 'origin' defined for it. This point is used either while referring to locations in 3D space, or while calculating the coordinates of pre-defined points with respect to the system.

It is a pretty well-known concept that there is no absolute notion of location or orientation in space. Any given coordinate system defines a unique 'perspective' of quantifying positions and directions. Therefore, even if we assume that all systems deal with the same units of measurement, the expression of vectorial and scalar quantities differs according to the coordinate system a certain observer deals with.

Consider two points P and Q in space. Assuming units to be common throughtout, the distance between these points remains the same regardless of the coordinate system in which the measurements are being made. However, the 3-D coordinates of each of the two points, as well as the position vector of any of the points with respect to the other, do not. In fact, these two quantities don't make sense at all, unless they are being measured keeping in mind a certain location and orientation of the measurer (essentially the coordinate system).

Therefore, it is quite clear that the orientation and location (of the origin) of a coordinate system define the way different quantities will be expressed with respect to it. Neither of the two properties can be measured on an absolute scale, but rather with respect to another coordinate system. The orientation of one system with respect to another is measured using the rotation matrix, while the relative position can be quantified via the position vector of one system's origin with respect to the other.

Fields

A **field** is a vector or scalar quantity that can be specified everywhere in space as a function of position (Note that in general a field may also be dependent on time and other custom variables). Since we only deal with 3D spaces in this module, a field is defined as a function of the x, y and z coordinates corresponding to a location in the coordinate system. Here, x, y and z act as scalar variables defining the position of a general point.

For example, temperature in 3 dimensional space (a temperature field) can be written as $T(x, y, z)$ – a scalar function of the position. An example of a scalar field in electromagnetism is the electric potential.

In a similar manner, a vector field can be defined as a vectorial function of the location (x, y, z) of any point in space.

For instance, every point on the earth may be considered to be in the gravitational force field of the earth. We may specify the field by the magnitude and the direction of acceleration due to gravity (i.e. force per unit mass) $\vec{g}(x, y, z)$ at every point in space.

To give an example from electromagnetism, consider an electric potential of form $2x^2y$, a scalar field in 3D space. The corresponding conservative electric field can be computed as the gradient of the electric potential function, and expressed as $4xy\hat{\mathbf{i}} + 2x^2\hat{\mathbf{j}}$. The magnitude of this electric field can in turn be expressed as a scalar field of the form $\sqrt{4x^4 + 16x^2y^2}$.

Basic Implementation details

Coordinate Systems and Vectors

Currently, *sympy.vector* (page 1425) is able to deal with the Cartesian (also called rectangular), spherical and other curvilinear coordinate systems.

A 3D Cartesian coordinate system can be initialized in *sympy.vector* (page 1425) as

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
```

The string parameter to the constructor denotes the name assigned to the system, and will primarily be used for printing purposes.

Once a coordinate system (in essence, a `CoordSys3D` instance) has been defined, we can access the orthonormal unit vectors (i.e. the $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$ and $\hat{\mathbf{k}}$ vectors) and coordinate variables/base scalars (i.e. the $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ variables) corresponding to it. We will talk about coordinate variables in the later sections.

The basis vectors for the X, Y and Z axes can be accessed using the `i`, `j` and `k` properties respectively.

```
>>> N.i
N.i
>>> type(N.i)
<class 'sympy.vector.vector.BaseVector'>
```

As seen above, the basis vectors are all instances of a class called `BaseVector`.

When a `BaseVector` is multiplied by a scalar (essentially any SymPy `Expr`), we get a `VectorMul` - the product of a base vector and a scalar.

```
>>> 3*N.i
3*N.i
>>> type(3*N.i)
<class 'sympy.vector.vector.VectorMul'>
```

Addition of `VectorMul` and `BaseVectors` gives rise to formation of `VectorAdd` - except for special cases, ofcourse.

```
>>> v = 2*N.i + N.j
>>> type(v)
<class 'sympy.vector.vector.VectorAdd'>
>>> v - N.j
2*N.i
>>> type(v - N.j)
<class 'sympy.vector.vector.VectorMul'>
```

What about a zero vector? It can be accessed using the `zero` attribute assigned to class `Vector`. Since the notion of a zero vector remains the same regardless of the coordinate system in consideration, we use `Vector.zero` wherever such a quantity is required.

```
>>> from sympy.vector import Vector
>>> Vector.zero
0
>>> type(Vector.zero)
<class 'sympy.vector.vector.VectorZero'>
>>> N.i + Vector.zero
N.i
>>> Vector.zero == 2*Vector.zero
True
```

All the classes shown above - `BaseVector`, `VectorMul`, `VectorAdd` and `VectorZero` are subclasses of `Vector`.

You should never have to instantiate objects of any of the subclasses of `Vector`. Using the `BaseVector` instances assigned to a `CoordSys3D` instance and (if needed) `Vector.zero` as building blocks, any sort of vectorial expression can be constructed with the basic mathematical operators `+`, `-`, `*`. and `/`.

```
>>> v = N.i - 2*N.j
>>> v/3
1/3*N.i + (-2/3)*N.j
>>> v + N.k
N.i + (-2)*N.j + N.k
>>> Vector.zero/2
0
>>> (v/3)*4
4/3*N.i + (-8/3)*N.j
```

In addition to the elementary mathematical operations, the vector operations of `dot` and `cross` can also be performed on `Vector`.

```
>>> v1 = 2*N.i + 3*N.j - N.k
>>> v2 = N.i - 4*N.j + N.k
>>> v1.dot(v2)
-11
>>> v1.cross(v2)
(-1)*N.i + (-3)*N.j + (-11)*N.k
>>> v2.cross(v1)
N.i + 3*N.j + 11*N.k
```

The `&` and `^` operators have been overloaded for the `dot` and `cross` methods respectively.

```
>>> v1 & v2
-11
>>> v1 ^ v2
(-1)*N.i + (-3)*N.j + (-11)*N.k
```

However, this is not the recommended way of performing these operations. Using the original methods makes the code clearer and easier to follow.

In addition to these operations, it is also possible to compute the outer products of `Vector` instances in *`sympy.vector`* (page 1425). More on that in a little bit.

SymPy operations on Vectors

The SymPy operations of `simplify`, `trigsimp`, `diff`, and `factor` work on `Vector` objects, with the standard SymPy API.

In essence, the methods work on the measure numbers(The coefficients of the basis vectors) present in the provided vectorial expression.

```
>>> from sympy.abc import a, b, c
>>> from sympy import sin, cos, trigsimp, diff
>>> v = (a*b + a*c + b**2 + b*c)*N.i + N.j
>>> v.factor()
((a + b)*(b + c))*N.i + N.j
>>> v = (sin(a)**2 + cos(a)**2)*N.i - (2*cos(b)**2 - 1)*N.k
>>> trigsimp(v)
N.i + (-cos(2*b))*N.k
>>> v.simplify()
N.i + (-cos(2*b))*N.k
>>> diff(v, b)
(4*sin(b)*cos(b))*N.k
>>> from sympy import Derivative
>>> Derivative(v, b).doit()
(4*sin(b)*cos(b))*N.k
```

`Integral` also works with `Vector` instances, similar to `Derivative`.

```
>>> from sympy import Integral
>>> v1 = a*N.i + sin(a)*N.j - N.k
>>> Integral(v1, a)
(Integral(a, a))*N.i + (Integral(sin(a), a))*N.j + (Integral(-1, a))*N.k
>>> Integral(v1, a).doit()
a**2/2*N.i + (-cos(a))*N.j + (-a)*N.k
```

Points

As mentioned before, every coordinate system corresponds to a unique origin point. Points, in general, have been implemented in *sympy.vector* (page 1425) in the form of the `Point` class.

To access the origin of system, use the `origin` property of the `CoordSys3D` class.

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N.origin
N.origin
>>> type(N.origin)
<class 'sympy.vector.point.Point'>
```

You can instantiate new points in space using the `locate_new` method of `Point`. The arguments include the name(string) of the new `Point`, and its position vector with respect to the 'parent' `Point`.

```
>>> from sympy.abc import a, b, c
>>> P = N.origin.locate_new('P', a*N.i + b*N.j + c*N.k)
>>> Q = P.locate_new('Q', -b*N.j)
```

Like `Vector`, a user never has to expressly instantiate an object of `Point`. This is because any location in space (albeit relative) can be pointed at by using the `origin` of a `CoordSys3D` as the reference, and then using `locate_new` on it and subsequent `Point` instances.

The position vector of a `Point` with respect to another `Point` can be computed using the `position_wrt` method.

```
>>> P.position_wrt(Q)
b*N.j
>>> Q.position_wrt(N.origin)
a*N.i + c*N.k
```

Additionally, it is possible to obtain the X, Y and Z coordinates of a `Point` with respect to a `CoordSys3D` in the form of a tuple. This is done using the `express_coordinates` method.

```
>>> Q.express_coordinates(N)
(a, 0, c)
```

Dyadics

A dyadic, or dyadic tensor, is a second-order tensor formed by the juxtaposition of pairs of vectors. Therefore, the outer products of vectors give rise to the formation of dyadics. Dyadic tensors have been implemented in *sympy.vector* (page 1425) in the `Dyadic` class.

Once again, you never have to instantiate objects of `Dyadic`. The outer products of vectors can be computed using the `outer` method of `Vector`. The `|` operator has been overloaded for `outer`.

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N.i.outer(N.j)
(N.i|N.j)
>>> N.i|N.j
(N.i|N.j)
```

Similar to `Vector`, `Dyadic` also has subsequent subclasses like `BaseDyadic`, `DyadicMul`, `DyadicAdd`. As with `Vector`, a zero dyadic can be accessed from `Dyadic.zero`.

All basic mathematical operations work with `Dyadic` too.

```
>>> dyad = N.i.outer(N.k)
>>> dyad*3
3*(N.i|N.k)
>>> dyad - dyad
0
>>> dyad + 2*(N.j|N.i)
(N.i|N.k) + 2*(N.j|N.i)
```

`dot` and `cross` also work among `Dyadic` instances as well as between a `Dyadic` and `Vector` (and also vice versa) - as per the respective mathematical definitions. As with `Vector`, `&` and

^ have been overloaded for dot and cross.

```
>>> d = N.i.outer(N.j)
>>> d.dot(N.j|N.j)
(N.i|N.j)
>>> d.dot(N.i)
0
>>> d.dot(N.j)
N.i
>>> N.i.dot(d)
N.j
>>> N.k ^ d
(N.j|N.j)
```

More about Coordinate Systems

We will now look at how we can initialize new coordinate systems in *sympy.vector* (page 1425), transformed in user-defined ways with respect to already-existing systems.

Locating new systems

We already know that the origin property of a CoordSys3D corresponds to the Point instance denoting its origin reference point.

Consider a coordinate system N. Suppose we want to define a new system M, whose origin is located at $\mathbf{3\hat{i}} + \mathbf{4\hat{j}} + \mathbf{5\hat{k}}$ from N's origin. In other words, the coordinates of M's origin from N's perspective happen to be $(3, 4, 5)$. Moreover, this would also mean that the coordinates of N's origin with respect to M would be $(-3, -4, -5)$.

This can be achieved programmatically as follows -

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> M = N.locate_new('M', 3*N.i + 4*N.j + 5*N.k)
>>> M.position_wrt(N)
3*N.i + 4*N.j + 5*N.k
>>> N.origin.express_coordinates(M)
(-3, -4, -5)
```

It is worth noting that M's orientation is the same as that of N. This means that the rotation matrix of :math: N with respect to M, and also vice versa, is equal to the identity matrix of dimensions 3x3. The locate_new method initializes a CoordSys3D that is only translated in space, not re-oriented, relative to the 'parent' system.

Orienting new systems

Similar to 'locating' new systems, *sympy.vector* (page 1425) also allows for initialization of new CoordSys3D instances that are oriented in user-defined ways with respect to existing systems.

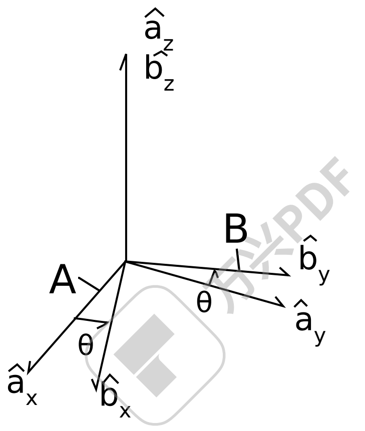
Suppose you have a coordinate system A.

```
>>> from sympy.vector import CoordSys3D
>>> A = CoordSys3D('A')
```

You want to initialize a new coordinate system B, that is rotated with respect to A's Z-axis by an angle θ.

```
>>> from sympy import Symbol
>>> theta = Symbol('theta')
```

The orientation is shown in the diagram below:



There are two ways to achieve this.

Using a method of CoordSys3D directly

This is the easiest, cleanest, and hence the recommended way of doing it.

```
>>> B = A.orient_new_axis('B', theta, A.k)
```

This initializes B with the required orientation information with respect to A.

CoordSys3D provides the following direct orientation methods in its API-

1. orient_new_axis
2. orient_new_body
3. orient_new_space
4. orient_new_quaternion

Please look at the CoordSys3D class API given in the docs of this module, to know their functionality and required arguments in detail.

Using Orienter(s) and the orient_new method

You would first have to initialize an `AxisOrienter` instance for storing the rotation information.

```
>>> from sympy.vector import AxisOrienter
>>> axis_orienter = AxisOrienter(theta, A.k)
```

And then apply it using the `orient_new` method, to obtain B.

```
>>> B = A.orient_new('B', axis_orienter)
```

`orient_new` also lets you orient new systems using multiple `Orienter` instances, provided in an iterable. The rotations/orientations are applied to the new system in the order the `Orienter` instances appear in the iterable.

```
>>> from sympy.vector import BodyOrienter
>>> from sympy.abc import a, b, c
>>> body_orienter = BodyOrienter(a, b, c, 'XYZ')
>>> C = A.orient_new('C', (axis_orienter, body_orienter))
```

The *`sympy.vector`* (page 1425) API provides the following four `Orienter` classes for orientation purposes-

1. `AxisOrienter`
2. `BodyOrienter`
3. `SpaceOrienter`
4. `QuaternionOrienter`

Please refer to the API of the respective classes in the docs of this module to know more.

In each of the above examples, the origin of the new coordinate system coincides with the origin of the 'parent' system.

```
>>> B.position_wrt(A)
0
```

To compute the rotation matrix of any coordinate system with respect to another one, use the `rotation_matrix` method.

```
>>> B = A.orient_new_axis('B', a, A.k)
>>> B.rotation_matrix(A)
Matrix([
[ cos(a), sin(a), 0],
[-sin(a), cos(a), 0],
[      0,      0, 1]])
>>> B.rotation_matrix(B)
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

Orienting AND Locating new systems

What if you want to initialize a new system that is not only oriented in a pre-defined way, but also translated with respect to the parent?

Each of the `orient_new_<method of orientation>` methods, as well as the `orient_new` method, support a `location` keyword argument.

If a `Vector` is supplied as the value for this `kwarg`, the new system's origin is automatically defined to be located at that position vector with respect to the parent coordinate system.

Thus, the orientation methods also act as methods to support orientation+ location of the new systems.

```
>>> C = A.orient_new_axis('C', a, A.k, location=2*A.j)
>>> C.position_wrt(A)
2*A.j
>>> from sympy.vector import express
>>> express(A.position_wrt(C), C)
(-2*sin(a))*C.i + (-2*cos(a))*C.j
```

More on the `express` function in a bit.

Transforming new system

The most general way of creating user-defined system is to use `transformation` parameter in `CoordSys3D`. Here we can define any transformation equations. If we are interested in some typical curvilinear coordinate system different that Cartesian, we can also use some predefined ones. It could be also possible to translate or rotate system by setting appropriate transformation equations.

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import sin, cos
>>> A = CoordSys3D('A', transformation='spherical')
>>> B = CoordSys3D('A', transformation=lambda x,y,z: (x*sin(y), x*cos(y), z))
```

In `CoordSys3D` is also dedicated method, `create_new` which works similarly to methods like `locate_new`, `orient_new_axis` etc.

```
>>> from sympy.vector import CoordSys3D
>>> A = CoordSys3D('A')
>>> B = A.create_new('B', transformation='spherical')
```

Expression of quantities in different coordinate systems

Vectors and Dyadics

As mentioned earlier, the same vector attains different expressions in different coordinate systems. In general, the same is true for scalar expressions and dyadic tensors.

sympy.vector (page 1425) supports the expression of vector/scalar quantities in different coordinate systems using the `express` function.

For purposes of this section, assume the following initializations-

```
>>> from sympy.vector import CoordSys3D, express
>>> from sympy.abc import a, b, c
>>> N = CoordSys3D('N')
>>> M = N.orient_new_axis('M', a, N.k)
```

`Vector` instances can be expressed in user defined systems using `express`.

```
>>> v1 = N.i + N.j + N.k
>>> express(v1, M)
(sin(a) + cos(a))*M.i + (-sin(a) + cos(a))*M.j + M.k
>>> v2 = N.i + M.j
>>> express(v2, N)
(1 - sin(a))*N.i + (cos(a))*N.j
```

Apart from `Vector` instances, `express` also supports reexpression of scalars (general SymPy `Expr`) and `Dyadic` objects.

`express` also accepts a second coordinate system for re-expressing `Dyadic` instances.

```
>>> d = 2*(M.i | N.j) + 3* (M.j | N.k)
>>> express(d, M)
(2*sin(a))*(M.i|M.i) + (2*cos(a))*(M.i|M.j) + 3*(M.j|M.k)
>>> express(d, M, N)
2*(M.i|N.j) + 3*(M.j|N.k)
```

Coordinate Variables

The location of a coordinate system's origin does not affect the re-expression of `BaseVector` instances. However, it does affect the way `BaseScalar` instances are expressed in different systems.

`BaseScalar` instances, are coordinate 'symbols' meant to denote the variables used in the definition of vector/scalar fields in *sympy.vector* (page 1425).

For example, consider the scalar field $\mathbf{T_N}(\mathbf{x}, \mathbf{y}, \mathbf{z}) = \mathbf{x} + \mathbf{y} + \mathbf{z}$ defined in system N. Thus, at a point with coordinates (a, b, c), the value of the field would be $a + b + c$. Now consider system R, whose origin is located at $(1, 2, 3)$ with respect to N (no change of orientation). A point with coordinates (a, b, c) in R has coordinates $(a + 1, b + 2, c + 3)$ in N. Therefore, the expression for $\mathbf{T_N}$ in R becomes $\mathbf{T_R}(x, y, z) = x + y + z + 6$.

Coordinate variables, if present in a vector/scalar/dyadic expression, can also be re-expressed in a given coordinate system, by setting the `variables` keyword argument of `express` to `True`.

The above mentioned example, done programmatically, would look like this -

```
>>> R = N.locate_new('R', N.i + 2*N.j + 3*N.k)
>>> T_N = N.x + N.y + N.z
>>> express(T_N, R, variables=True)
R.x + R.y + R.z + 6
```

Other expression-dependent methods

The `to_matrix` method of `Vector` and `express_coordinates` method of `Point` also return different results depending on the coordinate system being provided.

```
>>> P = R.origin.locate_new('P', a*R.i + b*R.j + c*R.k)
>>> P.express_coordinates(N)
(a + 1, b + 2, c + 3)
>>> P.express_coordinates(R)
(a, b, c)
>>> v = N.i + N.j + N.k
>>> v.to_matrix(M)
Matrix([
[ sin(a) + cos(a)],
[-sin(a) + cos(a)],
[               1]])
>>> v.to_matrix(N)
Matrix([
[1],
[1],
[1]])
```

Scalar and Vector Field Functionality

Implementation in sympy.vector

Scalar and vector fields

In *sympy.vector* (page 1425), every `CoordSys3D` instance is assigned basis vectors corresponding to the X, Y and Z axes. These can be accessed using the properties named `i`, `j` and `k` respectively. Hence, to define a vector **v** of the form $3\mathbf{\hat{i}} + 4\mathbf{\hat{j}} + 5\mathbf{\hat{k}}$ with respect to a given frame **R**, you would do

```
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> v = 3*R.i + 4*R.j + 5*R.k
```

Vector math and basic calculus operations with respect to vectors have already been elaborated upon in the earlier section of this module's documentation.

On the other hand, base scalars (or coordinate variables) are implemented in a special class called `BaseScalar`, and are assigned to every coordinate system, one for each axis from X, Y and Z. These coordinate variables are used to form the expressions of vector or scalar fields in 3D space. For a system `R`, the X, Y and Z `BaseScalars` instances can be accessed using the `R.x`, `R.y` and `R.z` expressions respectively.

Therefore, to generate the expression for the aforementioned electric potential field $2x^2y$, you would have to do

```
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> electric_potential = 2*R.x**2*R.y
```

(continues on next page)