pyttb Documentation

Danny Dunlavy, Nick Johnson

Jul 06, 2023

CONTENTS

1 Functionality **3**

2 Python API **5**
2.1 Python API 5

3 How to Cite **73**
3.1 BibTex Entries: Tensor Toolbox for Python 73

4 Contact **77**

5 Indices and tables **79**

Python Module Index **81**

Index **83**

Tensors (also known as multidimensional arrays or N-way arrays) are used in a variety of applications ranging from chemometrics to network analysis.

- This is open source software. Please see LICENSE for the terms of the license (2-clause BSD).
- For more information or for feedback on this project, please *contact us*.

CHAPTER

ONE

FUNCTIONALITY

pyttb provides the following classes and functions for manipulating dense, sparse, and structured tensors, along with algorithms for computing low-rank tensor models.

- Tensor Classes

  pyttb supports multiple tensor types, including dense and sparse, as well as specially structured tensors, such as the Krusal format (stored as factor matrices).

- Algorithms

  CP methods such as alternating least squares, direct optimization, and weighted optimization (for missing data). Also alternative decompositions such as Poisson Tensor Factorization via alternating Poisson regression.

CHAPTER

TWO

PYTHON API

2.1 Python API

2.1.1 Tensor Classes

pyttb.ktensor

class pyttb.**ktensor**

Bases: object

KTENSOR Class for Kruskal tensors (decomposed).

Contains the following data members:

`weights`: `numpy.ndarray` vector containing the weights of the rank-1 tensors defined by the outer products of the column vectors of the factor_matrices.

`factor_matrices`: `list` of `numpy.ndarray`. The length of the list is equal to the number of dimensions of the tensor. The shape of the ith element of the list is (n_i, r), where n_i is the length dimension i and r is the rank of the tensor (as well as the length of the weights vector).

Although the constructor *__init__()* can be used to create an empty *pyttb.ktensor*, there are several class methods that can be used to create an instance of this class:

- *from_data()*
- *from_tensor_type()*
- *from_factor_matrices()*
- *from_function()*
- *from_vector()*

Examples

For all examples listed below, the following module imports are assumed:

```
>>> import pyttb as ttb
>>> import numpy as np
```

classmethod from_data(*weights*, **factor_matrices*)

Construct a *pyttb.ktensor* from weights and factor matrices.

The length of the list or the number of arguments specified by *factor_matrices* must equal the length of *weights*. See *pyttb.ktensor* for parameter descriptions.

Parameters

- **weights** (numpy.ndarray, required)
- **factor_matrices** (list of numpy.ndarray or variable number of numpy.ndarray, required)

Returns

pyttb.ktensor

Examples

Create a *pyttb.ktensor* from weights and a list of factor matrices:

```
>>> weights = np.array([1., 2.])
>>> fm0 = np.array([[1., 2.], [3., 4.]])
>>> fm1 = np.array([[5., 6.], [7., 8.]])
>>> K = ttb.ktensor.from_data(weights, [fm0, fm1])
>>> print(K)
ktensor of shape 2 x 2
weights=[1. 2.]
factor_matrices[0] =
[[1. 2.]
 [3. 4.]]
factor_matrices[1] =
[[5. 6.]
 [7. 8.]]
```

Create a *pyttb.ktensor* from weights and factor matrices passed as arguments:

```
>>> K = ttb.ktensor.from_data(weights, fm0, fm1)
>>> print(K)
ktensor of shape 2 x 2
weights=[1. 2.]
factor_matrices[0] =
[[1. 2.]
 [3. 4.]]
factor_matrices[1] =
[[5. 6.]
 [7. 8.]]
```

classmethod from_tensor_type(*source*) → *ktensor*

Construct a *pyttb.ktensor* from another *pyttb.ktensor*. A deep copy of the data from the input *pyttb.ktensor* is used for the new *pyttb.ktensor*.

Parameters

source (*pyttb.ktensor*, required)

Returns

pyttb.ktensor

Examples

Create an instance of a *pyttb.ktensor*:

```
>>> fm0 = np.array([[1., 2.], [3., 4.]])
>>> fm1 = np.array([[5., 6.], [7., 8.]])
>>> factor_matrices = [fm0, fm1]
>>> K_source = ttb.ktensor.from_factor_matrices(factor_matrices)
>>> print(K_source)
ktensor of shape 2 x 2
weights=[1. 1.]
factor_matrices[0] =
[[1. 2.]
 [3. 4.]]
factor_matrices[1] =
[[5. 6.]
 [7. 8.]]
```

Create another instance of a *pyttb.ktensor* from the original one above:

```
>>> K = ttb.ktensor.from_tensor_type(K_source)
>>> print(K)
ktensor of shape 2 x 2
weights=[1. 1.]
factor_matrices[0] =
[[1. 2.]
 [3. 4.]]
factor_matrices[1] =
[[5. 6.]
 [7. 8.]]
```

See also *pyttb.ktensor.copy()*

classmethod from_factor_matrices(**factor_matrices*)

Construct a *pyttb.ktensor* from factor matrices. The weights of the returned *pyttb.ktensor* will all be equal to 1.

Parameters

factor_matrices (`list` of `numpy.ndarray` or variable number of `numpy.ndarray`, required) – The number of columns of each of the factor matrices must be the same.

Returns

pyttb.ktensor

Examples

Create a *pyttb.ktensor* from a `list` of factor matrices:

```
>>> fm0 = np.array([[1., 2.], [3., 4.]])
>>> fm1 = np.array([[5., 6.], [7., 8.]])
>>> factor_matrices = [fm0, fm1]
>>> K = ttb.ktensor.from_factor_matrices(factor_matrices)
>>> print(K)
ktensor of shape 2 x 2
```

(continues on next page)

(continued from previous page)

```
weights=[1. 1.]
factor_matrices[0] =
[[1. 2.]
 [3. 4.]]
factor_matrices[1] =
[[5. 6.]
 [7. 8.]]
```

Create a *pyttb.ktensor* from factor matrices passed as arguments:

```
>>> K = ttb.ktensor.from_factor_matrices(fm0, fm1)
>>> print(K)
ktensor of shape 2 x 2
weights=[1. 1.]
factor_matrices[0] =
[[1. 2.]
 [3. 4.]]
factor_matrices[1] =
[[5. 6.]
 [7. 8.]]
```

classmethod from_function(*fun*, *shape*, *num_components*)

Construct a *pyttb.ktensor* whose factor matrix entries are set using a function. The weights of the returned *pyttb.ktensor* will all be equal to 1.

Parameters

- **fun** (*function, required*) – A function that can accept a shape (i.e., `tuple` of dimension sizes) and return a `numpy.ndarray` of that shape. Example functions include *numpy.random.random_sample*, *numpy,zeros*, *numpy.ones*.
- **shape** (`tuple`, required)
- **num_components** (*int, required*)

Returns

pyttb.ktensor

Examples

Create a *pyttb.ktensor* with entries of the factor matrices taken from a uniform random distribution:

```
>>> np.random.seed(1)
>>> K = ttb.ktensor.from_function(np.random.random_sample, (2, 3, 4), 2)
>>> print(K)
ktensor of shape 2 x 3 x 4
weights=[1. 1.]
factor_matrices[0] =
[[4.1702...e-01 7.2032...e-01]
 [1.1437...e-04 3.0233...e-01]]
factor_matrices[1] =
[[0.1467... 0.0923...]
 [0.1862... 0.3455...]
 [0.3967... 0.5388...]]
```

(continues on next page)

(continued from previous page)

```
factor_matrices[2] =
[[0.4191... 0.6852...]
 [0.2044... 0.8781...]
 [0.0273... 0.6704...]
 [0.4173...  0.5586...]]
```

Create a *pyttb.ktensor* with entries equal to 1:

```
>>> K = ttb.ktensor.from_function(np.ones, (2, 3, 4), 2)
>>> print(K)
ktensor of shape 2 x 3 x 4
weights=[1. 1.]
factor_matrices[0] =
[[1. 1.]
 [1. 1.]]
factor_matrices[1] =
[[1. 1.]
 [1. 1.]
 [1. 1.]]
factor_matrices[2] =
[[1. 1.]
 [1. 1.]
 [1. 1.]
 [1. 1.]]
```

Create a *pyttb.ktensor* with entries equal to 0:

```
>>> K = ttb.ktensor.from_function(np.zeros, (2, 3, 4), 2)
>>> print(K)
ktensor of shape 2 x 3 x 4
weights=[1. 1.]
factor_matrices[0] =
[[0. 0.]
 [0. 0.]]
factor_matrices[1] =
[[0. 0.]
 [0. 0.]
 [0. 0.]]
factor_matrices[2] =
[[0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]
```

classmethod from_vector(*data*, *shape*, *contains_weights*)

Construct a *pyttb.ktensor* from a vector (given as a numpy.ndarray) and shape (given as a numpy.ndarray). The rank of the *pyttb.ktensor* is inferred from the shape and length of the vector.

Parameters

- **data** (numpy.ndarray, required) – Vector containing either elements of the factor matrices (when `contains_weights`==False) or elements of the weights and factor matrices (when `contains_weights`==True). When both the elements of the weights and the factor_matrices are present, the weights come first and the columns of the factor matrices

come next.

- **shape** (numpy.ndarray, required) – Vector containing the shape of the tensor (i.e., lengths of the dimensions).
- **contains_weights** (*bool, required*) – Flag to specify whether or not *data* contains weights. If False, all weights are set to 1.

Returns

pyttb.ktensor

Examples

Create a *pyttb.ktensor* from a vector containing only elements of the factor matrices:

```
>>> rank = 2
>>> shape = np.array([2, 3, 4])
>>> data = np.arange(1, rank*sum(shape)+1).astype(float)
>>> K = ttb.ktensor.from_vector(data[:], shape, False)
>>> print(K)
ktensor of shape 2 x 3 x 4
weights=[1. 1.]
factor_matrices[0] =
[[1. 3.]
 [2. 4.]]
factor_matrices[1] =
[[ 5.  8.]
 [ 6.  9.]
 [ 7. 10.]]
factor_matrices[2] =
[[11. 15.]
 [12. 16.]
 [13. 17.]
 [14. 18.]]
```

Create a *pyttb.ktensor* from a vector containing elements of both the weights and the factor matrices:

```
>>> weights = 2 * np.ones(rank).astype(float)
>>> weights_and_data = np.concatenate((weights, data), axis=0)
>>> K = ttb.ktensor.from_vector(weights_and_data[:], shape, True)
>>> print(K)
ktensor of shape 2 x 3 x 4
weights=[2. 2.]
factor_matrices[0] =
[[1. 3.]
 [2. 4.]]
factor_matrices[1] =
[[ 5.  8.]
 [ 6.  9.]
 [ 7. 10.]]
factor_matrices[2] =
[[11. 15.]
 [12. 16.]
 [13. 17.]
 [14. 18.]]
```

arrange(*weight_factor=None*, *permutation=None*)

Arrange the rank-1 components of a *pyttb.ktensor* in place. If *permutation* is passed, the columns of *self.factor_matrices* are arranged using the provided permutation, so you must make a copy before calling this method if you want to store the original *pyttb.ktensor*. If *weight_factor* is passed, then the values in *self.weights* are absorbed into *self.factor_matrices[weight_factor]*. If no parameters are passed, then the columns of *self.factor_matrices* are normalized and then permuted such that the resulting *self.weights* are sorted by magnitude, greatest to least. Passing both parameters leads to an error.

Parameters

- **weight_factor** (*int, optional*) – The index of the factor matrix that the weights will be absorbed into.
- **permutation** (tuple, list, or numpy.ndarray, optional) – The new order of the components of the *pyttb.ktensor* into which to permute. The permutation must be of length equal to the number of components of the *pyttb.ktensor*, *self.ncomponents* and must be a permutation of [0,…,`self.ncomponents`-1].

Examples

Create the initial *pyttb.ktensor*:

```
>>> weights = np.array([1., 2.])
>>> fm0 = np.array([[1., 2.], [3., 4.]])
>>> fm1 = np.array([[5., 6.], [7., 8.]])
>>> K = ttb.ktensor.from_data(weights, [fm0, fm1])
>>> print(K)
ktensor of shape 2 x 2
weights=[1. 2.]
factor_matrices[0] =
[[1. 2.]
 [3. 4.]]
factor_matrices[1] =
[[5. 6.]
 [7. 8.]]
```

Arrange the columns of the factor matrices using a permutation:

```
>>> p = [1,0]
>>> K.arrange(permutation=p)
>>> print(K)
ktensor of shape 2 x 2
weights=[2. 1.]
factor_matrices[0] =
[[2. 1.]
 [4. 3.]]
factor_matrices[1] =
[[6. 5.]
 [8. 7.]]
```

Normalize and permute columns such that *weights* are sorted in decreasing order:

```
>>> K.arrange()
>>> print(K)
```

(continues on next page)

(continued from previous page)

```
ktensor of shape 2 x 2
weights=[89.4427... 27.2029...]
factor_matrices[0] =
[[0.4472... 0.3162...]
 [0.8944... 0.9486...]]
factor_matrices[1] =
[[0.6... 0.5812...]
 [0.8... 0.8137...]]
```

Absorb the weights into the second factor:

```
>>> K.arrange(weight_factor=1)
>>> print(K)
ktensor of shape 2 x 2
weights=[1. 1.]
factor_matrices[0] =
[[0.4472... 0.3162...]
 [0.8944... 0.9486...]]
factor_matrices[1] =
[[53.6656... 15.8113...]
 [71.5541... 22.1359...]]
```

copy()

Make a deep copy of a *pyttb.ktensor*.

Returns

pyttb.ktensor

Examples

Create a random *pyttb.ktensor* with weights of 1:

```
>>> np.random.seed(1)
>>> K = ttb.ktensor.from_function(np.random.random_sample, (2, 3, 4), 2)
>>> print(K)
ktensor of shape 2 x 3 x 4
weights=[1. 1.]
factor_matrices[0] =
[[4.1702...e-01 7.2032...e-01]
 [1.1437...e-04 3.0233...e-01]]
factor_matrices[1] =
[[0.1467... 0.0923...]
 [0.1862... 0.3455...]
 [0.3967... 0.5388...]]
factor_matrices[2] =
[[0.4191... 0.6852...]
 [0.2044... 0.8781...]
 [0.0273... 0.6704...]
 [0.4173... 0.5586...]]
```

Create a copy of the *pyttb.ktensor* and change the weights:

```
>>> K2 = K.copy()
>>> K2.weights = np.array([2., 3.])
>>> print(K2)
ktensor of shape 2 x 3 x 4
weights=[2. 3.]
factor_matrices[0] =
[[4.1702...e-01 7.2032...e-01]
 [1.1437...e-04 3.023...e-01]]
factor_matrices[1] =
[[0.1467... 0.0923...]
 [0.1862... 0.3455...]
 [0.3967... 0.5388...]]
factor_matrices[2] =
[[0.4191... 0.6852...]
 [0.2044... 0.8781...]
 [0.0273... 0.6704...]
 [0.4173... 0.5586...]]
```

Show that the original *pyttb.ktensor* is unchanged:

```
>>> print(K)
ktensor of shape 2 x 3 x 4
weights=[1. 1.]
factor_matrices[0] =
[[4.1702...e-01 7.2032...e-01]
 [1.1437...e-04 3.0233...e-01]]
factor_matrices[1] =
[[0.1467... 0.0923...]
 [0.1862... 0.3455...]
 [0.3967... 0.5388...]]
factor_matrices[2] =
[[0.4191... 0.6852...]
 [0.2044... 0.8781...]
 [0.0273... 0.6704...]
 [0.4173... 0.5586...]]
```

double()

Convert *pyttb.ktensor* to numpy.ndarray.

Returns

numpy.ndarray

Examples

```
>>> weights = np.array([1., 2.])
>>> fm0 = np.array([[1., 2.], [3., 4.]])
>>> fm1 = np.array([[5., 6.], [7., 8.]])
>>> factor_matrices = [fm0, fm1]
>>> K = ttb.ktensor.from_data(weights, factor_matrices)
>>> K.double()
array([[29., 39.],
       [63., 85.]])
```

(continues on next page)

(continued from previous page)

```
>>> type(K.double())
<class 'numpy.ndarray'>
```

end(*k=None*)

Last index of indexing expression for *pyttb.ktensor*.

Parameters
k (*int, optional*) – dimension for subscripted indexing

Returns
int (*index*)

Examples

```
>>> K = ttb.ktensor.from_function(np.ones, (2, 3, 4), 2)
>>> print(K.end(2))
3
```

extract(*idx=None*)

Creates a new *pyttb.ktensor* with only the specified components.

Parameters
idx (int, `tuple`, `list`, `numpy.ndarray`, optional) – Index set of components to extract. It should be the case that *idx* is a subset of [0,...,`self.ncomponents`]. If this parameter is None or is empty, a copy of the *pyttb.ktensor* is returned.

Returns
pyttb.ktensor

Examples

Create a *pyttb.ktensor*:

```
>>> weights = np.array([1., 2.])
>>> fm0 = np.array([[1., 2.], [3., 4.]])
>>> fm1 = np.array([[5., 6.], [7., 8.]])
>>> K = ttb.ktensor.from_data(weights, [fm0, fm1])
>>> print(K)
ktensor of shape 2 x 2
weights=[1. 2.]
factor_matrices[0] =
[[1. 2.]
 [3. 4.]]
factor_matrices[1] =
[[5. 6.]
 [7. 8.]]
```

Create a new *pyttb.ktensor*, extracting only the second component from each factor of the original *pyttb.ktensor*:

```
>>> K.extract([1])
ktensor of shape 2 x 2
```

(continues on next page)

(continued from previous page)

```
weights=[2.]
factor_matrices[0] =
[[2.]
 [4.]]
factor_matrices[1] =
[[6.]
 [8.]]
```

fixsigns(*other=None*)

Change the elements of a *pyttb.ktensor* in place so that the largest magnitude entries for each column vector in each factor matrix are positive, provided that the sign on pairs of vectors in a rank-1 component can be flipped.

Parameters

other (*pyttb.ktensor*, optional) – If not None, returns a version of the *pyttb.ktensor* where some of the signs of the columns of the factor matrices have been flipped to better align with *other*. In not None, both *pyttb.ktensor* objects are first normalized (using *normalize()*).

Returns

pyttb.ktensor – The changes are made in place and a reference to the updated tensor is returned

Examples

Create a *pyttb.ktensor* with negative large magnitude entries:

```
>>> weights = np.array([1., 2.])
>>> fm0 = np.array([[1., 2.], [3., 4.]])
>>> fm1 = np.array([[5., 6.], [7., 8.]])
>>> K = ttb.ktensor.from_data(weights, [fm0, fm1])
>>> K.factor_matrices[0][1, 1] = -K.factor_matrices[0][1, 1]
>>> K.factor_matrices[1][1, 1] = -K.factor_matrices[1][1, 1]
>>> print(K)
ktensor of shape 2 x 2
weights=[1. 2.]
factor_matrices[0] =
[[ 1.  2.]
 [ 3. -4.]]
factor_matrices[1] =
[[ 5.  6.]
 [ 7. -8.]]
```

Fix the signs of the largest magnitude entries:

```
>>> print(K.fixsigns())
ktensor of shape 2 x 2
weights=[1. 2.]
factor_matrices[0] =
[[ 1. -2.]
 [ 3.  4.]]
factor_matrices[1] =
```

(continues on next page)

(continued from previous page)

```
[[ 5. -6.]
 [ 7.  8.]]
```

Fix the signs using another *pyttb.ktensor*:

```
>>> K = ttb.ktensor.from_data(weights, [fm0, fm1])
>>> K2 = K.copy()
>>> K2.factor_matrices[0][1, 1] = -K2.factor_matrices[0][1, 1]
>>> K2.factor_matrices[1][1, 1] = -K2.factor_matrices[1][1, 1]
>>> K = K.fixsigns(K2)
>>> print(K)
ktensor of shape 2 x 2
weights=[27.2029... 89.4427...]
factor_matrices[0] =
[[ 0.3162... -0.4472...]
 [ 0.9486... -0.8944...]]
factor_matrices[1] =
[[ 0.5812... -0.6...]
 [ 0.8137... -0.8...]]
```

full()

Convert a *pyttb.ktensor* to a *pyttb.tensor*.

Returns
pyttb.tensor

Examples

```
>>> weights = np.array([1., 2.])
>>> fm0 = np.array([[1., 2.], [3., 4.]])
>>> fm1 = np.array([[5., 6.], [7., 8.]])
>>> K = ttb.ktensor.from_data(weights, [fm0, fm1])
>>> print(K)
ktensor of shape 2 x 2
weights=[1. 2.]
factor_matrices[0] =
[[1. 2.]
 [3. 4.]]
factor_matrices[1] =
[[5. 6.]
 [7. 8.]]
>>> print(K.full())
tensor of shape 2 x 2
data[:, :] =
[[29. 39.]
 [63. 85.]]
```

innerprod(*other*)

Efficient inner product with a *pyttb.ktensor*.

Efficiently computes the inner product between two tensors, *self*
and *other*. If other is a *pyttb.ktensor*, the inner product is computed using inner products of the

factor matrices. Otherwise, the inner product is computed using the *ttv* (tensor times vector) of *other* with all of the columns of *self.factor_matrices*.

Parameters
other (*pyttb.ktensor*, *pyttb.sptensor*, *pyttb.tensor*, or *pyttb.ttensor*, required) – Tensor with which to compute the inner product.

Returns
float

Examples

```
>>> K = ttb.ktensor.from_function(np.ones, (2,3,4), 2)
>>> print(K.innerprod(K))
96.0
```

isequal(*other*)

Equal comparator for *pyttb.ktensor* objects.

Parameters
other (*pyttb.ktensor*, required) – *pyttb.ktensor* with which to compare.

Returns
bool

Examples

```
>>> K1 = ttb.ktensor.from_function(np.ones, (2,3,4), 2)
>>> weights = np.ones((2, 1))
>>> factor_matrices = [np.ones((2, 2)), np.ones((3, 2)), np.ones((4, 2))]
>>> K2 = ttb.ktensor.from_data(weights, factor_matrices)
>>> print(K1.isequal(K2))
True
```

issymmetric(*return_diffs=False*)

Returns True if the *pyttb.ktensor* is exactly symmetric for every permutation.

Parameters
return_diffs (*bool, optional*) – If True, returns the matrix of the norm of the differences between the factor matrices.

Returns

- *bool*
- numpy.ndarray, optional – Matrix of the norm of the differences between the factor matrices

Examples

Create a *pyttb.ktensor* that is symmetric and test if it is symmetric:

```
>>> K = ttb.ktensor.from_function(np.ones, (3, 3, 3), 2)
>>> print(K.issymmetric())
True
```

Create a *pyttb.ktensor* that is not symmetric and return the differences:

```
>>> weights = np.array([1., 2.])
>>> fm0 = np.array([[1., 2.], [3., 4.]])
>>> fm1 = np.array([[5., 6.], [7., 8.]])
>>> K2 = ttb.ktensor.from_data(weights, [fm0, fm1])
>>> issym, diffs = K2.issymmetric(return_diffs=True)
>>> print(diffs)
[[0. 8.]
 [0. 0.]]
```

mask(*W*)

Extract *pyttb.ktensor* values as specified by *W*, a *pyttb.tensor* or *pyttb.sptensor* containing only values of zeros (0) and ones (1). The values in the *pyttb.ktensor* corresponding to the indices for the ones (1) in *W* will be returned as a column vector.

Parameters

W (*pyttb.tensor* or *pyttb.sptensor*, required)

Returns

numpy.ndarray

Examples

Create a *pyttb.ktensor*:

```
>>> weights = np.array([1., 2.])
>>> fm0 = np.array([[1., 2.], [3., 4.]])
>>> fm1 = np.array([[5., 6.], [7., 8.]])
>>> K = ttb.ktensor.from_data(weights, [fm0, fm1])
```

Create a mask *pyttb.tensor* and extract the elements of the *pyttb.ktensor* using the mask:

```
>>> W = ttb.tensor.from_data(np.array([[0, 1], [1, 0]]))
>>> print(K.mask(W))
[[63.]
 [39.]]
```

mttkrp(*U*, *n*)

Matricized tensor times Khatri-Rao product for *pyttb.ktensor*.

Efficiently calculates the matrix product of the n-mode matricization of the *ktensor* with the Khatri-Rao product of all entries in U, a `list` of factor matrices, except the nth.

Parameters

- **U** (`list` of factor matrices, required)
- **n** (*int, required*) – Multiply by all modes except n.

Returns
numpy.ndarray

Examples

```
>>> K = ttb.ktensor.from_function(np.ones, (2, 3, 4), 2)
>>> U = [np.ones((2, 2)), np.ones((3, 2)), np.ones(((4, 2)))]
>>> print(K.mttkrp(U, 0))
[[24. 24.]
 [24. 24.]]
```

property ncomponents

Number of components in the *pyttb.ktensor* (i.e., number of columns in each factor matrix) of the *pyttb.ktensor*.

Returns
int

Examples

```
>>> K = ttb.ktensor.from_function(np.ones, (2, 3, 4), 2)
>>> print(K.ncomponents)
2
```

property ndims

Number of dimensions (i.e., number of factor matrices) of the *pyttb.ktensor*.

Returns
int

Examples

```
>>> K = ttb.ktensor.from_function(np.ones, (2, 3, 4), 2)
>>> print(K.ndims)
3
```

norm()

Compute the norm (i.e., square root of the sum of squares of entries) of a *pyttb.ktensor*.

Returns
int

Examples

```
>>> K = ttb.ktensor.from_function(np.ones, (2, 3, 4), 2)
>>> K.norm()
9.797958971132712
```

normalize(*weight_factor=None*, *sort=False*, *normtype=2*, *mode=None*)

Normalize the columns of the factor matrices of a *pyttb.ktensor* in place.

Parameters

- **weight_factor** (*{"all", int}, optional*) – Absorb the weights into one or more factors. If "all", absorb weight equally across all factors. If *int*, absorb weight into a single dimension (value must be in range(self.ndims)).
- **sort** (*bool, optional*) – Sort the columns in descending order of the weights.
- **normtype** (*{non-negative int, -1, -2, np.inf, -np.inf}, optional*) – Order of the norm (see `numpy.linalg.norm()` for possible values).
- **mode** (*int, optional*) – Index of factor matrix to normalize. A value of *None* means normalize all factor matrices.

Returns

pyttb.ktensor

Examples

```
>>> K = ttb.ktensor.from_function(np.ones, (2, 3, 4), 2)
>>> print(K.normalize())
ktensor of shape 2 x 3 x 4
weights=[4.898... 4.898...]
factor_matrices[0] =
[[0.7071... 0.7071...]
 [0.7071... 0.7071...]]
factor_matrices[1] =
[[0.5773... 0.5773...]
 [0.5773... 0.5773...]
 [0.5773... 0.5773...]]
factor_matrices[2] =
[[0.5 0.5]
 [0.5 0.5]
 [0.5 0.5]
 [0.5 0.5]]
```

nvecs(*n*, *r*, *flipsign=True*)

Compute the leading mode-n vectors for a *pyttb.ktensor*.

Computes the *r* leading eigenvectors of Xn*Xn.T (where Xn is the mode-*n* matricization/unfolding of self), which provides information about the mode-N fibers. In two-dimensions, the *r* leading mode-1 vectors are the same as the *r* left singular vectors and the *r* leading mode-2 vectors are the same as the *r* right singular vectors. By default, this method computes the top *r* eigenvectors of Xn*Xn.T.

Parameters

- **n** (*int, required*) – Mode for tensor matricization.

- **r** (*int, required*) – Number of eigenvectors to compute and use.
- **flipsign** (*bool, optional*) – If True, make each column's largest element positive.

Returns
numpy.ndarray

Examples

Compute single eigenvector for dimension 0:

```
>>> K = ttb.ktensor.from_function(np.ones, (2, 3, 4), 2)
>>> nvecs1 = K.nvecs(0, 1)
>>> print(nvecs1)
[[0.70710678...]
 [0.70710678...]]
```

Compute first 2 leading eigenvectors for dimension 0:

```
>>> nvecs2 = K.nvecs(0, 2)
>>> print(nvecs2)
[[ 0.70710678...  0.70710678...]
 [ 0.70710678... -0.70710678...]]
```

permute(*order*)

Permute *pyttb.ktensor* dimensions.

Rearranges the dimensions of a *pyttb.ktensor* so that they are in the order specified by *order*. The corresponding tensor has the same components as *self* but the order of the subscripts needed to access any particular element is rearranged as specified by *order*.

Parameters
order (numpy.ndarray) – Permutation of [0,...,self.ndimensions].

Returns
pyttb.ktensor

Examples

```
>>> weights = np.array([1., 2.])
>>> fm0 = np.array([[1., 2.], [3., 4.]])
>>> fm1 = np.array([[5., 6.], [7., 8.]])
>>> factor_matrices = [fm0, fm1]
>>> K = ttb.ktensor.from_data(weights, factor_matrices)
>>> print(K)
ktensor of shape 2 x 2
weights=[1. 2.]
factor_matrices[0] =
[[1. 2.]
 [3. 4.]]
factor_matrices[1] =
[[5. 6.]
 [7. 8.]]
```

Permute the order of the dimension so they are in reverse order:

```
>>> K1 = K.permute(np.array([1, 0]))
>>> print(K1)
ktensor of shape 2 x 2
weights=[1. 2.]
factor_matrices[0] =
[[5. 6.]
 [7. 8.]]
factor_matrices[1] =
[[1. 2.]
 [3. 4.]]
```

redistribute(*mode*)

Distribute weights of a *pyttb.ktensor* to the specified mode. The redistribution is performed in place.

Parameters
mode (*int*) – Must be value in [0,... self.ndims].

Example

Create a *pyttb.ktensor*:

```
>>> weights = np.array([1., 2.])
>>> fm0 = np.array([[1., 2.], [3., 4.]])
>>> fm1 = np.array([[5., 6.], [7., 8.]])
>>> factor_matrices = [fm0, fm1]
>>> K = ttb.ktensor.from_data(weights, factor_matrices)
>>> print(K)
ktensor of shape 2 x 2
weights=[1. 2.]
factor_matrices[0] =
[[1. 2.]
 [3. 4.]]
factor_matrices[1] =
[[5. 6.]
 [7. 8.]]
```

Distribute weights of that *pyttb.ktensor* to mode 0:

```
>>> K.redistribute(0)
>>> print(K)
ktensor of shape 2 x 2
weights=[1. 1.]
factor_matrices[0] =
[[1. 4.]
 [3. 8.]]
factor_matrices[1] =
[[5. 6.]
 [7. 8.]]
```

property shape

Shape of a *pyttb.ktensor*.

Returns the lengths of all dimensions of the *pyttb.ktensor*.

Returns
tuple

score(*other*, *weight_penalty=True*, *threshold=0.99*, *greedy=True*)

Checks if two *pyttb.ktensor* instances with the same shapes but potentially different number of components match except for permutation.

Matching is defined as follows. If *self* and *other* are single- component *pyttb.ktensor* instances that have been normalized so that their weights are *self.weights* and *other.weights*, and their factor matrices are single column vectors containing [a1,a2,...,an] and [b1,b2,...bn], rescpetively, then the score is defined as

score = penalty * (a1.T*b1) * (a2.T*b2) * ... * (an.T*bn),

where the penalty is defined by the weights such that

penalty = 1 - abs(self.weights - other.weights) / max(self.weights, other.weights).

The score of multi-component *pyttb.ktensor* instances is a normalized sum of the scores across the best permutation of the components of *self*. *self* can have more components than *other*; any extra components are ignored in terms of the matching score.

Parameters

- **other** (*pyttb.ktensor*, required) – *pyttb.ktensor* with which to match.
- **weight_penalty** (*bool, optional*) – Flag indicating whether or not to consider the weights in the calculations.
- **threshold** (*float, optional*) – Threshold specified in the formula above for determining a match.
- **greedy** (*bool, optional*) – Flag indicating whether or not to consider all possible matchings (exponentially expensive) or just do a greedy matching.

Returns

- *int* – Score (between 0 and 1).
- *pyttb.ktensor* – Copy of *self*, which has been normalized and permuted to best match *other*.
- *bool* – Flag indicating a match according to a user-specified threshold.
- *numpy.ndarray* – Permutation (i.e. array of indices of the modes of self) of the components of *self* that was used to best match *other*.

Examples

Create two *pyttb.ktensor* instances and compute the score between them:

```
>>> K = ttb.ktensor.from_data(np.array([2., 1., 3.]), np.ones((3,3)), np.
↪ones((4,3)), np.ones((5,3)))
>>> K2 = ttb.ktensor.from_data(np.array([2., 4.]), np.ones((3,2)), np.ones((4,
↪2)), np.ones((5,2)))
>>> score,Kperm,flag,perm = K.score(K2)
>>> print(score)
0.875
>>> print(perm)
[0 2 1]
```

Compute score without using weights:

```
>>> score,Kperm,flag,perm = K.score(K2,weight_penalty=False)
>>> print(score)
1.0
>>> print(perm)
[0 1 2]
```

symmetrize()

Symmetrize a *pyttb.ktensor* in all modes.

Symmetrize a *pyttb.ktensor* with respect to all modes so that the resulting *pyttb.ktensor* is symmetric with respect to any permutation of indices.

Returns

pyttb.ktensor

Examples

Create a *pyttb.ktensor*:

```
>>> weights = np.array([1., 2.])
>>> fm0 = np.array([[1., 2.], [3., 4.]])
>>> fm1 = np.array([[5., 6.], [7., 8.]])
>>> factor_matrices = [fm0, fm1]
>>> K = ttb.ktensor.from_data(weights, factor_matrices)
>>> print(K)
ktensor of shape 2 x 2
weights=[1. 2.]
factor_matrices[0] =
[[1. 2.]
 [3. 4.]]
factor_matrices[1] =
[[5. 6.]
 [7. 8.]]
```

Make the factor matrices of the *pyttb.ktensor* symmetric with respect to any permutation of the factor matrices:

```
>>> K1 = K.symmetrize()
>>> print(K1)
ktensor of shape 2 x 2
weights=[1. 1.]
factor_matrices[0] =
[[2.3404... 4.9519...]
 [4.5960... 8.0124...]]
factor_matrices[1] =
[[2.3404... 4.9519...]
 [4.5960... 8.0124...]]
```

tolist(*mode=None*)

Convert *pyttb.ktensor* to a list of factor matrices, evenly distributing the weights across factors. Optionally absorb the weights into a single mode.

Parameters

mode (*int, optional*) – Index of factor matrix to absorb all of the weights.

Returns
list of numpy.ndarray

Examples

Create a *pyttb.ktensor* of all ones:

```
>>> weights = np.array([1., 2.])
>>> fm0 = np.array([[1., 2.], [3., 4.]])
>>> fm1 = np.array([[5., 6.], [7., 8.]])
>>> factor_matrices = [fm0, fm1]
>>> K = ttb.ktensor.from_data(weights, factor_matrices)
>>> print(K)
ktensor of shape 2 x 2
weights=[1. 2.]
factor_matrices[0] =
[[1. 2.]
 [3. 4.]]
factor_matrices[1] =
[[5. 6.]
 [7. 8.]]
```

Spread weights equally to all factors and return list of factor matrices:

```
>>> fm_list = K.tolist()
>>> for fm in fm_list: print(fm)
[[1. 2.8284...]
 [3. 5.6568...]]
[[ 5. 8.4852...]
 [ 7. 11.313...]]
```

Shift weight to single factor matrix and return list of factor matrices:

```
>>> fm_list = K.tolist(0)
>>> for fm in fm_list: print(fm)
[[ 8.6023... 40. ]
 [25.8069... 80. ]]
[[0.5812... 0.6...]
 [0.8137... 0.8...]]
```

tovec(*include_weights=True*)

Convert *pyttb.ktensor* to column vector. Optionally include or exclude the weights.

Parameters
include_weights (*bool, optional*) – Flag to specify whether or not to include weights in output.

Returns
numpy.ndarray – The length of the column vector is (sum(self.shape)+1)*self.ncomponents. The vector contains the weights (if requested) stacked on top of each of the columns of the factor_matrices in order.

Examples

Create a *pyttb.ktensor* from a vector:

```
>>> rank = 2
>>> shape = np.array([2, 3, 4])
>>> data = np.arange(1, rank*sum(shape)+1)
>>> weights = 2 * np.ones(rank)
>>> weights_and_data = np.concatenate((weights, data), axis=0)
>>> K = ttb.ktensor.from_vector(weights_and_data[:], shape, True)
>>> print(K)
ktensor of shape 2 x 3 x 4
weights=[2. 2.]
factor_matrices[0] =
[[1. 3.]
 [2. 4.]]
factor_matrices[1] =
[[ 5.  8.]
 [ 6.  9.]
 [ 7. 10.]]
factor_matrices[2] =
[[11. 15.]
 [12. 16.]
 [13. 17.]
 [14. 18.]]
```

Create a *pyttb.ktensor* from a vector of data extracted from another *pyttb.ktensor*:

```
>>> K2 = ttb.ktensor.from_vector(K.tovec(), shape, True)
>>> print(K2)
ktensor of shape 2 x 3 x 4
weights=[2. 2.]
factor_matrices[0] =
[[1. 3.]
 [2. 4.]]
factor_matrices[1] =
[[ 5.  8.]
 [ 6.  9.]
 [ 7. 10.]]
factor_matrices[2] =
[[11. 15.]
 [12. 16.]
 [13. 17.]
 [14. 18.]]
```

ttv(*vector*, *dims=None*, *exclude_dims=None*)

Tensor times vector for a *pyttb.ktensor*.

Computes the product of a *pyttb.ktensor* with a vector (i.e., np.array). If *dims* is an integer, it specifies the dimension in the *pyttb.ktensor* along which the vector is multiplied. If the shape of the vector is = (I,1), then the length of dimension *dims* of the *pyttb.ktensor* must be I. Note that the number of dimensions of the returned *pyttb.ktensor* is 1 less than the dimension of the *pyttb.ktensor* used in the multiplication because dimension *dims* is removed.

If *vector* is a `list` of np.array instances, the *pyttb.ktensor* is multiplied with each vector in the list. The

products are computed sequentially along all dimensions (or modes) of the *pyttb.ktensor*, and thus the list must contain *self.ndims* vectors.

When *dims* is not None, compute the products along the dimensions specified by *dims*. In this case, the number of products can be less than *self.ndims* and the order of the sequence does not need to match the order of the dimensions in the *pyttb.ktensor*. Note that the number of vectors must match the number of dimensions provided, and the length of each vector must match the size of each dimension of the *pyttb.ktensor* specified in *dims*.

Parameters

- **vector** (numpy.ndarray or list[numpy.ndarray], required)
- **dims** (int, numpy.ndarray, optional)
- **exclude_dims**

Returns

float or *pyttb.ktensor* – The number of dimensions of the returned *pyttb.ktensor* is n-k, where n = self.ndims and k = number of vectors provided as input. If k == n, a scalar is returned.

Examples

Compute the product of a *pyttb.ktensor* and a single vector (results in a *pyttb.ktensor*):

```
>>> rank = 2
>>> shape = np.array([2, 3, 4])
>>> data = np.arange(1, rank*sum(shape)+1)
>>> weights = 2 * np.ones(rank)
>>> weights_and_data = np.concatenate((weights, data), axis=0)
>>> K = ttb.ktensor.from_vector(weights_and_data[:], shape, True)
>>> K0 = K.ttv(np.array([1, 1, 1]),dims=1) # compute along a single dimension
>>> print(K0)
ktensor of shape 2 x 4
weights=[36. 54.]
factor_matrices[0] =
[[1. 3.]
 [2. 4.]]
factor_matrices[1] =
[[11. 15.]
 [12. 16.]
 [13. 17.]
 [14. 18.]]
```

Compute the product of a *pyttb.ktensor* and a vector for each dimension (results in a *float*):

```
>>> vec2 = np.array([1, 1])
>>> vec3 = np.array([1, 1, 1])
>>> vec4 = np.array([1, 1, 1, 1])
>>> K1 = K.ttv([vec2, vec3, vec4])
>>> print(K1)
30348.0
```

Compute the product of a *pyttb.ktensor* and multiple vectors out of order (results in a *pyttb.ktensor*):

```
>>> K2 = K.ttv([vec4, vec3],np.array([2, 1]))
>>> print(K2)
ktensor of shape 2
weights=[1800. 3564.]
factor_matrices[0] =
[[1. 3.]
 [2. 4.]]
```

update(*modes*, *data*)

Updates a *pyttb.ktensor* in the specific dimensions with the values in *data* (in vector or matrix form). The value of *modes* must be a value in [-1,...,self.ndoms]. If the Further, the number of elements in *data* must equal self.shape[modes] * self.ncomponents. The update is performed in place.

Parameters

- **modes** (int or `list` of int, required) – List of dimensions to update; values must be in ascending order. If the first element of the list is -1, then update the weights. All other integer values values must be sorted and in [0,...,self.ndims].
- **data** (`numpy.ndarray`, required) – Data values to use in the update.
- **Results**
- **———-**
- **:class:`pyttb.ktensor`**

Examples

Create a *pyttb.ktensor* of all ones:

```
>>> K = ttb.ktensor.from_function(np.ones, (2, 3, 4), 2)
```

Create vectors for updating various factor matrices of the *pyttb.ktensor*:

```
>>> vec0 = 2 * np.ones(K.shape[0] * K.ncomponents)
>>> vec1 = 3 * np.ones(K.shape[1] * K.ncomponents)
>>> vec2 = 4 * np.ones(K.shape[2] * K.ncomponents)
```

Update a single factor matrix:

```
>>> K1 = K.copy()
>>> K1 = K1.update(0, vec0)
>>> print(K1)
ktensor of shape 2 x 3 x 4
weights=[1. 1.]
factor_matrices[0] =
[[2. 2.]
 [2. 2.]]
factor_matrices[1] =
[[1. 1.]
 [1. 1.]
 [1. 1.]]
factor_matrices[2] =
[[1. 1.]
```

(continues on next page)

(continued from previous page)

```
 [1. 1.]
 [1. 1.]
 [1. 1.]]
```

Update all factor matrices:

```
>>> K2 = K.copy()
>>> vec_all = np.concatenate((vec0, vec1, vec2))
>>> K2 = K2.update([0, 1, 2], vec_all)
>>> print(K2)
ktensor of shape 2 x 3 x 4
weights=[1. 1.]
factor_matrices[0] =
[[2. 2.]
 [2. 2.]]
factor_matrices[1] =
[[3. 3.]
 [3. 3.]
 [3. 3.]]
factor_matrices[2] =
[[4. 4.]
 [4. 4.]
 [4. 4.]
 [4. 4.]]
```

Update some but not all factor matrices:

```
>>> K3 = K.copy()
>>> vec_some = np.concatenate((vec0, vec2))
>>> K3 = K3.update([0, 2], vec_some)
>>> print(K3)
ktensor of shape 2 x 3 x 4
weights=[1. 1.]
factor_matrices[0] =
[[2. 2.]
 [2. 2.]]
factor_matrices[1] =
[[1. 1.]
 [1. 1.]
 [1. 1.]]
factor_matrices[2] =
[[4. 4.]
 [4. 4.]
 [4. 4.]
 [4. 4.]]
```

__add__(*other*)

Binary addition for *pyttb.ktensor*.

Parameters

other (*pyttb.ktensor*, required) – *pyttb.ktensor* to add to *self*.

Returns

pyttb.ktensor

__getitem__(*item*)

Subscripted reference for a *pyttb.ktensor*.

Subscripted reference is used to query the components of a *pyttb.ktensor*.

Parameters
item (*tuple(int) or int, required*)

Examples

```
>>> K = ttb.ktensor.from_function(np.ones, (2, 3, 4), 2)
>>> K.weights
array([1., 1.])
>>> K.factor_matrices
[array([[1., 1.],
       [1., 1.]]), array([[1., 1.],
       [1., 1.],
       [1., 1.]]), array([[1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.]])]
>>> K.factor_matrices[0]
array([[1., 1.],
       [1., 1.]])
>>> K[0]
array([[1., 1.],
       [1., 1.]])
>>> K[1, 2, 0]
2.0
>>> K[0][:, [0]]
array([[1.],
       [1.]])
```

__neg__()

Unary minus (negative) for *pyttb.ktensor* instances.

Returns
pyttb.ktensor

__pos__()

Unary plus (positive) for *pyttb.ktensor* instances.

Returns
pyttb.ktensor

__setitem__(*key*, *value*)

Subscripted assignment for *pyttb.ktensor*.

Subscripted assignment cannot be used to update individual elements of a *pyttb.ktensor*. You can update the weights vector or the factor matrices of a *pyttb.ktensor*.

Example

```
>>> K = ttb.ktensor.from_data(np.ones((4,1)), [np.random.random((2,4)), np.
↪random.random((3,4)), np.random.random((4,4))])
>>> K.weights = 2 * np.ones((4,1))
>>> K.factor_matrices[0] = np.zeros((2, 4))
>>> K.factor_matrices = [np.zeros((2, 4)), np.zeros((3, 4)), np.zeros((4, 4))]
>>> print(K)
ktensor of shape 2 x 3 x 4
weights=[[2.]
 [2.]
 [2.]
 [2.]]
factor_matrices[0] =
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
factor_matrices[1] =
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
factor_matrices[2] =
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

__sub__(*other*)

Binary subtraction for *pyttb.ktensor*.

Parameters
other (*pyttb.ktensor*)

Returns
pyttb.ktensor

__mul__(*other*)

Elementwise (including scalar) multiplication for *pyttb.ktensor* instances.

Parameters
other (*pyttb.tensor*, *pyttb.sptensor*, float, int)

Returns
pyttb.ktensor

__rmul__(*other*)

Elementwise (including scalar) multiplication for *pyttb.ktensor* instances.

Parameters
other (*pyttb.tensor*, *pyttb.sptensor*, float, int)

Returns
pyttb.ktensor

__repr__()

String representation of a *pyttb.ktensor*.

Returns
str

__str__()

String representation of a *pyttb.ktensor*.

Returns
str

__module__ = 'pyttb.ktensor'

pyttb.sptensor

Sparse Tensor Implementation

pyttb.sptensor.**tt_to_sparse_matrix**(*sptensorInstance:* sptensor, *mode: int*, *transpose: bool = False*) → coo_matrix

Helper function to unwrap sptensor into sparse matrix, should replace the core need for sptenmat

Parameters

- **sptensorInstance** (*sparse tensor to unwrap*)
- **mode** (*Mode around which to unwrap tensor*)
- **transpose** (*Whether or not to tranpose unwrapped tensor*)

Returns
spmatrix (*unwrapped tensor*)

pyttb.sptensor.**tt_from_sparse_matrix**(*spmatrix: coo_matrix*, *shape: Any*, *mode: int*, *idx: int*) → *sptensor*

Helper function to wrap sparse matrix into sptensor. Inverse of `pyttb.tt_to_sparse_matrix`

Parameters

- **spmatrix** (`Scipy.sparse.coo_matrix`)
- **mode** (*int*) – Mode around which tensor was unwrapped
- **idx** (*int*) – in {0,1}, idx of mode in spmatrix, s.b. 0 for tranpose=True

Returns
sptensorInstance (*pyttb.sptensor*)

class pyttb.sptensor.**sptensor**

Bases: `object`

SPTENSOR Class for sparse tensors.

__init__()

Create an empty sparse tensor

Returns
pyttb.sptensor

classmethod from_data(*subs: ndarray*, *vals: ndarray*, *shape: Tuple[int, ...]*) → *sptensor*

Construct an sptensor from fully defined SUB, VAL and SIZE matrices. This does no validation to optimize for speed when components are known. For default initializer with error checking see *from_aggregator()*.

Parameters

- **subs** (*location of non-zero entries*)
- **vals** (*values for non-zero entries*)

- **shape** (*shape of sparse tensor*)

Examples

Import required modules:

```
>>> import pyttb as ttb
>>> import numpy as np
```

Set up input data # Create sptensor with explicit data description

```
>>> subs = np.array([[1, 2], [1, 3]])
>>> vals = np.array([[6], [7]])
>>> shape = (4, 4, 4)
>>> K0 = ttb.sptensor.from_data(subs,vals, shape)
```

classmethod from_tensor_type(*source: Union[*sptensor, tensor, *coo_matrix]*) → *sptensor*

Contruct an `pyttb.sptensor` from compatible tensor types

Parameters
source (*Source tensor to create sptensor from*)

Returns
Generated Sparse Tensor

classmethod from_function(*function_handle: Callable[[Tuple[int, ...]], ndarray], shape: Tuple[int, ...], nonzeros: float*) → *sptensor*

Creates a sparse tensor of the specified shape with NZ nonzeros created from the specified function handle

Parameters

- **function_handle** (*function that accepts 2 arguments and generates*) – `numpy.ndarray` of length nonzeros
- **shape** (*tuple*)
- **nonzeros** (*int or float*)

Returns
Generated Sparse Tensor

classmethod from_aggregator(*subs: ndarray, vals: ndarray, shape: Optional[Tuple[int, ...]] = None, function_handle: Union[str, Callable[[Any], Union[float, ndarray]]] = 'sum'*) → *sptensor*

Construct an sptensor from fully defined SUB, VAL and shape matrices, after an aggregation is applied

Parameters

- **subs** (*location of non-zero entries*)
- **vals** (*values for non-zero entries*)
- **shape** (*shape of sparse tensor*)
- **function_handle** (*Aggregation function, or name of supported*) – aggregation function from numpy_groupies

Returns
Generated Sparse Tensor

Examples

```
>>> subs = np.array([[1, 2], [1, 3]])
>>> vals = np.array([[6], [7]])
>>> shape = np.array([4, 4])
>>> K0 = ttb.sptensor.from_aggregator(subs,vals)
>>> K1 = ttb.sptensor.from_aggregator(subs,vals,shape)
>>> function_handle = sum
>>> K2 = ttb.sptensor.from_aggregator(subs,vals,shape,function_handle)
```

allsubs() → ndarray

Generate all possible subscripts for sparse tensor

Returns

s (*All possible subscripts for sptensor*)

collapse(*dims: ~typing.Optional[~numpy.ndarray] = None, fun: ~typing.Callable[[~numpy.ndarray], ~typing.Union[float, ~numpy.ndarray]] = <function sum>*) → Union[float, ndarray, *sptensor*]

Collapse sparse tensor along specified dimensions.

Parameters

- **dims** (*Dimensions to collapse*)
- **fun** (*Method used to collapse dimensions*)

Returns

Collapsed value

Example

```
>>> subs = np.array([[1, 2], [1, 3]])
>>> vals = np.array([[1], [1]])
>>> shape = np.array([4, 4])
>>> X = ttb.sptensor.from_data(subs, vals, shape)
>>> X.collapse()
2
>>> X.collapse(np.arange(X.ndims), sum)
2
```

contract(*i: int*, *j: int*) → Union[ndarray, *sptensor*, *tensor*]

Contract tensor along two dimensions (array trace).

Parameters

- **i** (*First dimension*)
- **j** (*Second dimension*)

Returns

Contracted sptensor, converted to tensor if sufficiently dense

Example

```
>>> X = ttb.tensor.from_data(np.ones((2,2)))
>>> Y = sptensor.from_tensor_type(X)
>>> Y.contract(0, 1)
2.0
```

double() → ndarray

Convert sptensor to dense multidimensional array

elemfun(*function_handle: Callable[[ndarray], ndarray]*) → *sptensor*

Manipulate the non-zero elements of a sparse tensor

Parameters
function_handle (*Function that updates all values.*)

Returns
Updated sptensor

Example

```
>>> X = ttb.tensor.from_data(np.ones((2,2)))
>>> Y = sptensor.from_tensor_type(X)
>>> Z = Y.elemfun(lambda values: values*2)
>>> Z.isequal(Y*2)
True
```

end(*k: Optional[int] = None*) → int

Last index of indexing expression for sparse tensor

Parameters
k (*int Dimension for subscript indexing*)

extract(*searchsubs: ndarray*) → ndarray

Extract value for a sptensor.

Parameters
searchsubs (*subscripts to find in sptensor*)

See also:

__getitem__()

find() → Tuple[ndarray, ndarray]

FIND Find subscripts of nonzero elements in a sparse tensor.

Returns

- **subs** (*Subscripts of nonzero elements*)
- **vals** (*Values at corresponding subscripts*)

full() → *tensor*

FULL Convert a sparse tensor to a (dense) tensor.

innerprod(*other: Union[*sptensor, tensor, ktensor, ttensor*]*) → float

Efficient inner product with a sparse tensor

Parameters
other (*Other tensor to take innerproduct with*)

`isequal`(*other: Union[*sptensor, tensor*]*) → bool

Exact equality for sptensors

Parameters
other (*Other tensor to compare against*)

`logical_and`(*B: Union[float,* sptensor, tensor*]*) → *sptensor*

Logical and with self and another object

Parameters
B (*Other value to compare with*)

Returns
Indicator tensor

`logical_not`() → *sptensor*

Logical NOT for sptensors

Returns

- *Sparse tensor with all zero-values marked from original*
- *sparse tensor*

`logical_or`(*B: Union[float,* tensor*]*) → *tensor*

`logical_or`(*B:* sptensor) → *sptensor*

Logical OR for sptensor and another value

Returns
Indicator tensor

`logical_xor`(*other: Union[float,* tensor*]*) → *tensor*

`logical_xor`(*other:* sptensor) → *sptensor*

Logical XOR for sptensors

Parameters
other (*Other value to xor against*)

Returns
Indicator tensor

`mask`(*W:* sptensor) → ndarray

Extract values as specified by a mask tensor

Parameters
W (*Mask tensor*)

Returns
Extracted values

`mttkrp`(*U: Union[*ktensor, *List[ndarray]]*, *n: int*) → ndarray

Matricized tensor times Khatri-Rao product for sparse tensor.

Parameters

- **U** (*Matrices to create the Khatri-Rao product*)
- **n** (*Mode to matricize sptensor in*)

Returns
Matrix product

Examples

```
>>> matrix = np.ones((4, 4))
>>> subs = np.array([[1, 1, 1], [1, 1, 3], [2, 2, 2], [3, 3, 3]])
>>> vals = np.array([[0.5], [1.5], [2.5], [3.5]])
>>> shape = (4, 4, 4)
>>> sptensorInstance = sptensor.from_data(subs, vals, shape)
>>> sptensorInstance.mttkrp(np.array([matrix, matrix, matrix]), 0)
array([[0. , 0. , 0. , 0. ],
       [2. , 2. , 2. , 2. ],
       [2.5, 2.5, 2.5, 2.5],
       [3.5, 3.5, 3.5, 3.5]])
```

property ndims: `int`

NDIMS Number of dimensions of a sparse tensor.

property nnz: `int`

Number of nonzeros in sparse tensor

norm() → floating

Compute the Frobenius norm of a sparse tensor.

nvecs(*n: int*, *r: int*, *flipsign: bool = True*) → ndarray

Compute the leading mode-n vectors for a sparse tensor.

Parameters

- **n** (*Mode to unfold*)
- **r** (*Number of eigenvectors to compute*)
- **flipsign** (*Make each eigenvector's largest element positive*)

ones() → *sptensor*

Replace nonzero elements of sparse tensor with ones

permute(*order: ndarray*) → *sptensor*

Rearrange the dimensions of a sparse tensor

Parameters
order (*Updated order of dimensions*)

reshape(*new_shape: Tuple[int, ...]*, *old_modes: Optional[Union[ndarray, int]] = None*) → *sptensor*

Reshape specified modes of sparse tensor

Parameters

- **new_shape** (*tuple*)
- **old_modes** (`Numpy.ndarray`)

Returns
pyttb.sptensor

scale(*factor: ndarray*, *dims: Union[float, ndarray]*) → *sptensor*

Scale along specified dimensions for sparse tensors

Parameters

- **factor** (numpy.ndarray)
- **dims** (int or numpy.ndarray)

Returns
pyttb.sptensor

spmatrix() → coo_matrix

Converts a two-way sparse tensor to a sparse matrix in scipy.sparse.coo_matrix format

squeeze() → Union[*sptensor*, float]

Remove singleton dimensions from a sparse tensor

Returns
pyttb.sptensor or float if sptensor is only singleton dimensions

subdims(*region: Sequence[Union[int, np.ndarray, slice]]*) → np.ndarray

SUBDIMS Compute the locations of subscripts within a subdimension.

Parameters
region (numpy.ndarray or tuple denoting indexing) – Subset of total sptensor shape in which to find non-zero values

Returns
numpy.ndarray – Index into subs for non-zero values in region

Examples

```
>>> subs = np.array([[1, 1, 1], [1, 1, 3], [2, 2, 2], [3, 3, 3]])
>>> vals = np.array([[0.5], [1.5], [2.5], [3.5]])
>>> shape = (4, 4, 4)
>>> sp = sptensor.from_data(subs,vals,shape)
>>> region = [np.array([1]), np.array([1]), np.array([1,3])]
>>> loc = sp.subdims(region)
>>> print(loc)
[0 1]
>>> region = (1, 1, slice(None, None, None))
>>> loc = sp.subdims(region)
>>> print(loc)
[0 1]
```

ttv(*vector: Union[ndarray, List[ndarray]]*, *dims: Optional[Union[ndarray, int]] = None*, *exclude_dims: Optional[Union[ndarray, int]] = None*) → Union[*sptensor*, *tensor*]

Sparse tensor times vector

Parameters

- **vector** (*Vector(s) to multiply against*)
- **dims** (*Dimensions to multiply with vector(s)*)
- **exclude_dims** (*Use all dimensions but these*)

__getitem__(*item*)

Subscripted reference for a sparse tensor.

We can extract elements or subtensors from a sparse tensor in the following ways.

Case 1a: y = X(i1,i2,...,iN), where each in is an index, returns a scalar.

Case 1b: Y = X(R1,R2,...,RN), where one or more Rn is a range and the rest are indices, returns a sparse tensor. The elements are renumbered here as appropriate.

Case 2a: V = X(S) or V = X(S,'extract'), where S is a p x n array of subscripts, returns a vector of p values.

Case 2b: V = X(I) or V = X(I,'extract'), where I is a set of p linear indices, returns a vector of p values.

Any ambiguity results in executing the first valid case. This is particularily an issue if ndims(X)==1.

Parameters
item (tuple(int),tuple(slice),:class:*numpy.ndarray*)

Returns
numpy.ndarray or *pyttb.sptensor*

Examples

```
>>> subs = np.array([[3,3,3],[1,1,0],[1,2,1]])
>>> vals = np.array([3,5,1])
>>> shape = (4,4,4)
>>> X = sptensor.from_data(subs,vals,shape)
>>> _ = X[0,1,0] #<-- returns zero
>>> _ = X[3,3,3] #<-- returns 3
>>> _ = X[2:3,:,:] #<-- returns 1 x 4 x 4 sptensor
```

__setitem__(*key*, *value*)

Subscripted assignment for sparse tensor.

We can assign elements to a sptensor in three ways.

Case 1: X(R1,R2,...,RN) = Y, in which case we replace the rectangular subtensor (or single element) specified by the ranges R1,...,RN with Y. The right-hand-side can be a scalar or an sptensor.

Case 2: X(S) = V, where S is a p x n array of subscripts and V is a scalar value or a vector containing p values.

Linear indexing is not supported for sparse tensors.

Examples

X = sptensor([30 40 20]) <– Create an emtpy 30 x 40 x 20 sptensor X(30,40,20) = 7 <– Assign a single element to be 7 X([1,1,1;2,2,2]) = 1 <– Assign a list of elements to the same value X(11:20,11:20,11:20) = sptenrand([10,10,10],10) <– subtensor! X(31,41,21) = 7 <– grows the size of the tensor X(111:120,111:120,111:120) = sptenrand([10,10,10],10) <– grows X(1,1,1,1) = 4 <– increases the number of dimensions from 3 to 4

X = sptensor([30]) <– empty one-dimensional tensor X([4:6]) = 1 <– set subtensor to ones (does not increase dimension) X([10;12;14]) = (4:6)' <– set three elements X(31) = 7 <– grow the first dimension X(1,1) = 0 <– add a dimension, but no nonzeros

Note regarding singleton dimensions: It is not possible to do, for instance, X(1,1:10,1:10) = sptenrand([1 10 10],5). However, it is okay to do X(1,1:10,1:10) = squeeze(sptenrand([1 10 10],5)).

Parameters

- **key** (tuple(int),tuple(slice),:class:*numpy.ndarray*)
- **value** (int,float, numpy.ndarray, *pyttb.sptensor*)

__eq__(*other*)

Equal comparator for sptensors

Parameters
other (*compare equality of sptensor to other*)

Returns
pyttb.sptensor

__ne__(*other*)

Not equal comparator (~=) for sptensors

Parameters
other (*compare equality of sptensor to other*)

Returns
pyttb.sptensor

__sub__(*other*)

MINUS Binary subtraction for sparse tensors.

Parameters
other (*pyttb.tensor*, *pyttb.sptensor*)

Returns
pyttb.sptensor

__add__(*other*)

MINUS Binary addition for sparse tensors.

Parameters
other (*pyttb.tensor*, *pyttb.sptensor*)

Returns
pyttb.sptensor

__pos__()

Unary plus (+) for sptensors

Returns
pyttb.sptensor, copy of tensor

__neg__()

Unary minus (-) for sptensors

Returns
pyttb.sptensor, copy of tensor

__mul__(*other*)

Element wise multiplication (*) for sptensors

Parameters
other (*pyttb.sptensor*, *pyttb.tensor*, float, int)

Returns
pyttb.sptensor

__rmul__(*other*)

Element wise right multiplication (*) for sptensors

Parameters
other (*float, int*)

Returns
pyttb.sptensor

__le__(*other*)

Less than or equal (<=) for sptensor

Parameters
other (*pyttb.sptensor*, *pyttb.tensor*, float, int)

Returns
pyttb.sptensor

__lt__(*other*)

Less than (<) for sptensor

Parameters
other (*pyttb.sptensor*, *pyttb.tensor*, float, int)

Returns
pyttb.sptensor

__ge__(*other*)

Greater than or equal (>=) to for sptensor

Parameters
other (*pyttb.sptensor*, *pyttb.tensor*, float, int)

Returns
pyttb.sptensor

__gt__(*other*)

Greater than (>) to for sptensor

Parameters
other (*pyttb.sptensor*, *pyttb.tensor*, float, int)

Returns
pyttb.sptensor

__truediv__(*other*)

Division for sparse tensors (sptensor/other).

Parameters
other

__rtruediv__(*other*)

Right Division for sparse tensors (other/sptensor).

Parameters
other

__repr__()

String representation of a sparse tensor.

Returns
str – Contains the shape, subs and vals as strings on different lines.

__hash__ = None

__module__ = 'pyttb.sptensor'

__str__()

String representation of a sparse tensor.

Returns

str – Contains the shape, subs and vals as strings on different lines.

ttm(*matrices: Union[ndarray, List[ndarray]]*, *dims: Optional[Union[float, ndarray]] = None*, *exclude_dims: Optional[Union[float, ndarray]] = None*, *transpose: bool = False*)

Sparse tensor times matrix.

Parameters

- **matrices** (*A matrix or list of matrices*)
- **dims** (*Dimensions to multiply against*)
- **exclude_dims** (*Use all dimensions but these*)
- **transpose** (*Transpose matrices to be multiplied*)

pyttb.sptensor.**sptenrand**(*shape: Tuple[int, ...]*, *density: Optional[float] = None*, *nonzeros: Optional[float] = None*) → *sptensor*

Create sptensor with entries drawn from a uniform distribution on the unit interval

Parameters

- **shape** (*Shape of resulting tensor*)
- **density** (*Density of resulting sparse tensor*)
- **nonzeros** (*Number of nonzero entries in resulting sparse tensor*)

Returns

Constructed tensor

Example

```
>>> X = ttb.sptenrand((2,2), nonzeros=1)
>>> Y = ttb.sptenrand((2,2), density=0.25)
```

pyttb.sptensor.**sptendiag**(*elements: ndarray*, *shape: Optional[Tuple[int, ...]] = None*) → *sptensor*

Creates a sparse tensor with elements along super diagonal If provided shape is too small the tensor will be enlarged to accomodate

Parameters

- **elements** (*Elements to set along the diagonal*)
- **shape** (*Shape of resulting tensor*)

Returns

Constructed tensor

Example

```
>>> shape = (2,)
>>> values = np.ones(shape)
>>> X = ttb.sptendiag(values)
>>> Y = ttb.sptendiag(values, (2, 2))
>>> X.isequal(Y)
True
```

pyttb.tensor

Dense Tensor Implementation

class pyttb.tensor.**tensor**

Bases: object

TENSOR Class for dense tensors.

__init__()

TENSOR Create empty tensor.

data: ndarray

shape: Tuple

classmethod from_data(*data: ndarray*, *shape: Optional[Tuple[int, ...]] = None*) → *tensor*

Creates a tensor from explicit description. Note that 1D tensors (i.e., when len(shape)==1) contains a data array that follow the Numpy convention of being a row vector, which is different than in the Matlab Tensor Toolbox.

Parameters

- **data** (*Tensor source data*)
- **shape** (*Shape of resulting tensor if not the same as data shape*)

Returns

Constructed tensor

Example

```
>>> X = ttb.tensor.from_data(np.ones((2,2)))
>>> Y = ttb.tensor.from_data(np.ones((2,2)), shape=(4,1))
```

classmethod from_tensor_type(*source: Union[*sptensor, tensor, ktensor, tenmat*]*) → *tensor*

Converts other tensor types into a dense tensor

Parameters

source (*Tensor type to create dense tensor from*)

Returns

Constructed tensor

Example

```
>>> X = ttb.tensor.from_data(np.ones((2,2)))
>>> Y = ttb.tensor.from_tensor_type(X)
```

classmethod from_function(*function_handle: Callable[[Tuple[int, ...]], ndarray]*, *shape: Tuple[int, ...]*) → *tensor*

Creates a tensor from a function handle and size

Parameters

- **function_handle** (*Function to generate data to construct tensor*)
- **shape** (*Shape of resulting tensor*)

Returns

Constructed tensor

Example

```
>>> X = ttb.tensor.from_function(lambda a_shape: np.ones(a_shape), (2,2))
```

collapse(*dims: ~typing.Optional[~numpy.ndarray] = None*, *fun: ~typing.Callable[[~numpy.ndarray], ~typing.Union[float, ~numpy.ndarray]] = <function sum>*) → Union[float, ndarray, *tensor*]

Collapse tensor along specified dimensions.

Parameters

- **dims** (*Dimensions to collapse*)
- **fun** (*Method used to collapse dimensions*)

Returns

Collapsed value

Example

```
>>> X = ttb.tensor.from_data(np.ones((2,2)))
>>> X.collapse()
4.0
>>> X.collapse(np.arange(X.ndims), sum)
4.0
```

contract(*i: int*, *j: int*) → Union[ndarray, *tensor*]

Contract tensor along two dimensions (array trace).

Parameters

- **i** (*First dimension*)
- **j** (*Second dimension*)

Returns

Contracted tensor

Example

```
>>> X = ttb.tensor.from_data(np.ones((2,2)))
>>> X.contract(0, 1)
2.0
```

double() → ndarray

Convert tensor to an array of doubles

Returns
Copy of tensor data

Example

```
>>> X = ttb.tensor.from_data(np.ones((2,2)))
>>> X.double()
array([[1., 1.],
       [1., 1.]])
```

exp() → *tensor*

Exponential of the elements of tensor

Returns
Copy of tensor data element-wise raised to exponential

Examples

```
>>> tensor1 = ttb.tensor.from_data(np.array([[1, 2], [3, 4]]))
>>> tensor1.exp().data
array([[ 2.7182...,  7.3890... ],
       [20.0855..., 54.5981...]])
```

end(*k: Optional[int] = None*) → int

Last index of indexing expression for tensor

Parameters
k (*dimension for subscripted indexing*)

Examples

```
>>> X = ttb.tensor.from_data(np.ones((2,2)))
>>> X.end()  # linear indexing
3
>>> X.end(0)
1
```

find() → Tuple[ndarray, ndarray]

FIND Find subscripts of nonzero elements in a tensor.

S, V = FIND(X) returns the subscripts of the nonzero values in X and a column vector of the values.

Examples

```
>>> X = ttb.tensor.from_data(np.zeros((3,4,2)))
>>> larger_entries = X > 0.5
>>> subs, vals = larger_entries.find()
```

See also:

TENSOR, TENSOR

Returns
Subscripts and values for non-zero entries

full() → *tensor*

Convert dense tensor to dense tensor.

Returns
Deep copy

innerprod(*other: Union[*tensor, sptensor, ktensor*]*) → float

Efficient inner product with a tensor

Parameters
other (*Tensor type to take an innerproduct with*)

Examples

```
>>> tensor1 = ttb.tensor.from_data(np.array([[1, 2], [3, 4]]))
>>> tensor1.innerprod(tensor1)
30
```

isequal(*other: Union[*tensor, sptensor*]*) → bool

Exact equality for tensors

Parameters
other (*Tensor to compare against*)

Examples

```
>>> X = ttb.tensor.from_data(np.ones((2,2)))
>>> Y = ttb.tensor.from_data(np.zeros((2,2)))
>>> X.isequal(Y)
False
```

issymmetric(*grps: Optional[ndarray] = None*, *version: Optional[Any] = None*, *return_details: bool = False*) → Union[bool, Tuple[bool, ndarray, ndarray]]

Determine if a dense tensor is symmetric in specified modes.

Parameters

- **grps** (*Modes to check for symmetry*)
- **version** (*Flag*) – Any non-None value will call the non-default old version
- **return_details** (*Flag to return symmetry details in addition to bool*)

Returns
If symmetric in modes, optionally all differences and permutations

Examples

```
>>> X = ttb.tensor.from_data(np.ones((2,2)))
>>> X.issymmetric()
True
>>> X.issymmetric(grps=np.arange(X.ndims))
True
>>> is_sym, diffs, perms =             X.issymmetric(grps=np.arange(X.ndims),␣
↪version=1, return_details=True)
>>> print(f"Tensor is symmetric: {is_sym}")
Tensor is symmetric: True
>>> print(f"Differences in modes: {diffs}")
Differences in modes: [[0.]
 [0.]]
>>> print(f"Permutations: {perms}")
Permutations: [[0. 1.]
 [1. 0.]]
```

logical_and(*B: Union[float,* tensor*]*) → *tensor*

Logical and for tensors

Parameters
B (*Value to and against self*)

Examples

```
>>> X = ttb.tensor.from_data(np.ones((2,2), dtype=bool))
>>> X.logical_and(X).collapse()  # All true
4
```

logical_not() → *tensor*

Logical Not For Tensors

Returns
Negated tensor

Examples

```
>>> X = ttb.tensor.from_data(np.ones((2,2), dtype=bool))
>>> X.logical_not().collapse()  # All false
0
```

logical_or(*other: Union[float,* tensor*]*) → *tensor*

Logical or for tensors

Parameters
other (*Value to perform or against*)

Examples

```
>>> X = ttb.tensor.from_data(np.ones((2,2), dtype=bool))
>>> X.logical_or(X.logical_not()).collapse()  # All true
4
```

logical_xor(*other: Union[float,* tensor*]*) → *tensor*

Logical xor for tensors

Parameters
other (*Value to perform xor against*)

Examples

```
>>> X = ttb.tensor.from_data(np.ones((2,2), dtype=bool))
>>> X.logical_xor(X.logical_not()).collapse()  # All true
4
```

mask(*W:* tensor) → ndarray

Extract non-zero values at locations specified by mask tensor

Parameters
W (*Mask tensor*)

Returns
Extracted values

Examples

```
>>> W = ttb.tensor.from_data(np.ones((2,2)))
>>> tensor1 = ttb.tensor.from_data(np.array([[1, 2], [3, 4]]))
>>> tensor1.mask(W)
array([1, 3, 2, 4])
```

mttkrp(*U: Union[*ktensor*, List[ndarray]]*, *n: int*) → ndarray

Matricized tensor times Khatri-Rao product

Parameters

- **U** (*Matrices to create the Khatri-Rao product*)
- **n** (*Mode to matricize tensor in*)

Returns
Matrix product

Example

```
>>> tensor1 = ttb.tensor.from_data(np.ones((2,2,2)))
>>> matrices = [np.ones((2,2))] * 3
>>> tensor1.mttkrp(matrices, 2)
array([[4., 4.],
       [4., 4.]])
```

property ndims: *int*

Return the number of dimensions of a tensor

Examples

```
>>> X = ttb.tensor.from_data(np.ones((2,2)))
>>> X.ndims
2
```

property nnz: *int*

Number of non-zero elements in tensor

Examples

```
>>> X = ttb.tensor.from_data(np.ones((2,2)))
>>> X.nnz
4
```

norm() → floating

Frobenius Norm of Tensor

Examples

```
>>> X = ttb.tensor.from_data(np.ones((2,2)))
>>> X.norm()
2.0
```

nvecs(*n: int*, *r: int*, *flipsign: bool = True*) → ndarray

Compute the leading mode-n eigenvectors for a tensor

Parameters

- **n** (*Mode to unfold*)
- **r** (*Number of eigenvectors to compute*)
- **flipsign** (*Make each eigenvector's largest element positive*)

Examples

```
>>> tensor1 = ttb.tensor.from_data(np.array([[1, 2], [3, 4]]))
>>> tensor1.nvecs(0,1)
array([[0.4045...],
       [0.9145...]])
>>> tensor1.nvecs(0,2)
array([[ 0.4045...,  0.9145...],
       [ 0.9145..., -0.4045...]])
```

permute(*order: ndarray*) → *tensor*

Permute tensor dimensions.

Parameters
order (*New order of tensor dimensions*)

Returns
Updated tensor with shapeNew == shapePrevious[order]

Examples

```
>>> X = ttb.tensor.from_data(np.ones((2,2)))
>>> Y = X.permute(np.array((1,0)))
>>> X.isequal(Y)
True
```

reshape(*shape: Tuple[int, ...]*) → *tensor*

Reshapes a tensor

Parameters
shape (*New shape*)

Examples

```
>>> X = ttb.tensor.from_data(np.ones((2,2)))
>>> Y = X.reshape((4,1))
>>> Y.shape
(4, 1)
```

squeeze() → Union[*tensor*, ndarray, float]

Removes singleton dimensions from a tensor

Returns
Tensor or scalar if all dims squeezed

Examples

```
>>> tensor1 = ttb.tensor.from_data(np.array([[[4]]]))
>>> tensor1.squeeze()
4
>>> tensor2 = ttb.tensor.from_data(np.array([[1, 2, 3]]))
>>> tensor2.squeeze().data
array([1, 2, 3])
```

symmetrize(*grps: Optional[ndarray] = None*, *version: Optional[Any] = None*) → *tensor*

Symmetrize a tensor in the specified modes .. rubric:: Notes

It is *the same or less* work to just call X = symmetrize(X) then to first check if X is symmetric and then symmetrize it, even if X is already symmetric.

Parameters

- **grps** (*Modes to check for symmetry*)
- **version** (*Any non-None value will call the non-default old version*)

ttm(*matrix: Union[ndarray, List[ndarray]]*, *dims: Optional[Union[float, ndarray]] = None*, *exclude_dims: Optional[Union[ndarray, int]] = None*, *transpose: bool = False*) → *tensor*

Tensor times matrix

Parameters

- **matrix** (*Matrix or matrices to multiple by*)
- **dims** (*Dimensions to multiply against*)
- **exclude_dims** (*Use all dimensions but these*)
- **transpose** (*Transpose matrices during multiplication*)

ttt(*other:* tensor, *selfdims: Optional[Union[ndarray, int]] = None*, *otherdims: Optional[Union[ndarray, int]] = None*) → *tensor*

Tensor multiplication (tensor times tensor)

Parameters

- **other** (*Tensor to multiply by*)
- **selfdims** (*Dimensions to contract this tensor by for multiplication*)
- **otherdims** (*Dimensions to contract other tensor by for multiplication*)

ttv(*vector: Union[ndarray, List[ndarray]]*, *dims: Optional[Union[ndarray, int]] = None*, *exclude_dims: Optional[Union[ndarray, int]] = None*) → *tensor*

Tensor times vector

Parameters

- **vector** (*Vector(s) to multiply against*)
- **dims** (*Dimensions to multiply with vector(s)*)
- **exclude_dims** (*Use all dimensions but these*)

ttsv(*vector: Union[ndarray, List[ndarray]]*, *skip_dim: Optional[int] = None*, *version: Optional[int] = None*) → Union[ndarray, *tensor*]

Tensor times same vector in multiple modes

Parameters

- **vector** (*Vector(s) to multiply against*)
- **skip_dim** (*Multiply tensor by vector in all dims except [0, skip_dim]*)

__setitem__(*key*, *value*)

SUBSASGN Subscripted assignment for a tensor.

We can assign elements to a tensor in three ways.

Case 1: X(R1,R2,...,RN) = Y, in which case we replace the rectangular subtensor (or single element) specified by the ranges R1,...,RN with Y. The right-hand-side can be a scalar, a tensor, or an MDA.

Case 2a: X(S) = V, where S is a p x n array of subscripts and V is a scalar or a vector containing p values.

Case 2b: X(I) = V, where I is a set of p linear indices and V is a scalar or a vector containing p values. Resize is not allowed in this case.

Examples X = tensor(rand(3,4,2)) X(1:2,1:2,1) = ones(2,2) <– replaces subtensor X([1 1 1;1 1 2]) = [5;7] <– replaces two elements X([1;13]) = [5;7] <– does the same thing X(1,1,2:3) = 1 <– grows tensor X(1,1,4) = 1 %<- grows the size of the tensor

__getitem__(*item*)

SUBSREF Subscripted reference for tensors.

We can extract elements or subtensors from a tensor in the following ways.

Case 1a: y = X(i1,i2,...,iN), where each in is an index, returns a scalar.

Case 1b: Y = X(R1,R2,...,RN), where one or more Rn is a range and the rest are indices, returns a sparse tensor.

Case 2a: V = X(S) or V = X(S,'extract'), where S is a p x n array of subscripts, returns a vector of p values.

Case 2b: V = X(I) or V = X(I,'extract'), where I is a set of p linear indices, returns a vector of p values.

Any ambiguity results in executing the first valid case. This is particularly an issue if ndims(X)==1.

Examples X = tensor(rand(3,4,2,1),[3 4 2 1]); X.data <– returns multidimensional array X.size <– returns size X(1,1,1,1) <– produces a scalar X(1,1,1,:) <– produces a tensor of order 1 and size 1 X(:,1,1,:) <– produces a tensor of size 3 x 1 X(1:2,[2 4],1,:) <– produces a tensor of size 2 x 2 x 1 X(1:2,[2 4],1,1) <– produces a tensor of size 2 x 2 X([1,1,1,1;3,4,2,1]) <– returns a vector of length 2 X = tensor(rand(10,1),10); X([1:6]') <– extracts a subtensor X([1:6]','extract') <– extracts a vector of 6 elements

Returns
pyttb.tensor or numpy.ndarray

__eq__(*other*)

Equal for tensors

Parameters
other (*pyttb.tensor*, float, int)

Returns
pyttb.tensor

__ne__(*other*)

Not equal (!=) for tensors

Parameters
other (*pyttb.tensor*, float, int)

Returns
pyttb.tensor

__ge__(*other*)

Greater than or equal (>=) for tensors

Parameters
other (*pyttb.tensor*, float, int)

Returns
pyttb.tensor

__le__(*other*)

Less than or equal (<=) for tensors

Parameters
other (*pyttb.tensor*, float, int)

Returns
pyttb.tensor

__gt__(*other*)

Greater than (>) for tensors

Parameters
other (*pyttb.tensor*, float, int)

Returns
pyttb.tensor

__lt__(*other*)

Less than (<) for tensors

Parameters
other (*pyttb.tensor*, float, int)

Returns
pyttb.tensor

__sub__(*other*)

Binary subtraction (-) for tensors

Parameters
other (*pyttb.tensor*, float, int)

Returns
pyttb.tensor

__add__(*other*)

Binary addition (+) for tensors

Parameters
other (*pyttb.tensor*, float, int)

Returns
pyttb.tensor

__radd__(*other*)

Right binary addition (+) for tensors

Parameters
other (*pyttb.tensor*, float, int)

Returns
pyttb.tensor

__pow__(*power*)

Element Wise Power (**) for tensors

Parameters
other (*pyttb.tensor*, float, int)

Returns
pyttb.tensor

__mul__(*other*)

Element wise multiplication (*) for tensors, self*other

Parameters
other (*pyttb.tensor*, float, int)

Returns
pyttb.tensor

__rmul__(*other*)

Element wise right multiplication (*) for tensors, other*self

Parameters
other (*pyttb.tensor*, float, int)

Returns
pyttb.tensor

__truediv__(*other*)

Element wise left division (/) for tensors, self/other

Parameters
other (*pyttb.tensor*, float, int)

Returns
pyttb.tensor

__rtruediv__(*other*)

Element wise right division (/) for tensors, other/self

Parameters
other (*pyttb.tensor*, float, int)

Returns
pyttb.tensor

__pos__()

Unary plus (+) for tensors

Returns
pyttb.tensor – copy of tensor

__neg__()

Unary minus (-) for tensors

Returns
pyttb.tensor – copy of tensor

__repr__()

String representation of a tensor.

Returns
str – Contains the shape and data as strings on different lines.

__str__()

String representation of a tensor.

Returns

str – Contains the shape and data as strings on different lines.

__annotations__ = {'data': 'np.ndarray', 'shape': 'Tuple'}

__hash__ = None

__module__ = 'pyttb.tensor'

pyttb.tensor.**tenones**(*shape: Tuple[int, ...]*) → *tensor*

Creates a tensor of all ones

Parameters

shape (*Shape of resulting tensor*)

Returns

Constructed tensor

Example

```
>>> X = ttb.tenones((2,2))
```

pyttb.tensor.**tenzeros**(*shape: Tuple[int, ...]*) → *tensor*

Creates a tensor of all zeros

Parameters

shape (*Shape of resulting tensor*)

Returns

Constructed tensor

Example

```
>>> X = ttb.tenzeros((2,2))
```

pyttb.tensor.**tenrand**(*shape: Tuple[int, ...]*) → *tensor*

Creates a tensor with entries drawn from a uniform distribution on the unit interval

Parameters

shape (*Shape of resulting tensor*)

Returns

Constructed tensor

Example

```
>>> X = ttb.tenrand((2,2))
```

pyttb.tensor.**tendiag**(*elements: ndarray*, *shape: Optional[Tuple[int, ...]] = None*) → *tensor*

Creates a tensor with elements along super diagonal If provided shape is too small the tensor will be enlarged to accomodate

Parameters

- **elements** (*Elements to set along the diagonal*)
- **shape** (*Shape of resulting tensor*)

Returns

Constructed tensor

Example

```
>>> shape = (2,)
>>> values = np.ones(shape)
>>> X = ttb.tendiag(values)
>>> Y = ttb.tendiag(values, (2, 2))
>>> X.isequal(Y)
True
```

pyttb.ttensor

class pyttb.ttensor.**ttensor**

Bases: `object`

TTENSOR Class for Tucker tensors (decomposed).

__init__()

Create an empty decomposed tucker tensor

Returns

pyttb.ttensor

classmethod from_data(*core*, *factors*)

Construct an ttensor from fully defined core tensor and factor matrices.

Parameters

- **core** (:class: *ttb.tensor*)
- **factors** (`list(numpy.ndarray)`)

Returns

pyttb.ttensor

Examples

Import required modules:

```
>>> import pyttb as ttb
>>> import numpy as np
```

Set up input data # Create ttensor with explicit data description

```
>>> core_values = np.ones((2,2,2))
>>> core = ttb.tensor.from_data(core_values)
>>> factors = [np.ones((1,2))] * len(core_values.shape)
>>> K0 = ttb.ttensor.from_data(core, factors)
```

classmethod from_tensor_type(*source*)

Converts other tensor types into a ttensor

Parameters
source (*pyttb.ttensor*)

Returns
pyttb.ttensor

property shape

Shape of the tensor this deconstruction represents.

Returns
tuple(int)

__repr__()

String representation of a tucker tensor.

Returns
str – Contains the core, and factor matrices as strings on different lines.

__str__()

String representation of a tucker tensor.

Returns
str – Contains the core, and factor matrices as strings on different lines.

full()

Convert a ttensor to a (dense) tensor.

Returns
pyttb.tensor

double()

Convert ttensor to an array of doubles

Returns
numpy.ndarray – copy of tensor data

property ndims

Number of dimensions of a ttensor.

Returns
int – Number of dimensions of ttensor

isequal(*other*)

Component equality for ttensors

Parameters
other (*pyttb.ttensor*)

Returns
bool (*True if ttensors decompositions are identical, false otherwise*)

__pos__()

Unary plus (+) for ttensors. Does nothing.

Returns
pyttb.ttensor, copy of tensor

__neg__()

Unary minus (-) for ttensors

Returns
pyttb.ttensor, copy of tensor

innerprod(*other*)

Efficient inner product with a ttensor

Parameters

- **other** (*pyttb.tensor*, *pyttb.sptensor*, *pyttb.ktensor*,)
- **:class:`pyttb.ttensor`**

Returns
float

__mul__(*other*)

Element wise multiplication (*) for ttensors (only scalars supported)

Parameters
other (*float, int*)

Returns
pyttb.ttensor

__rmul__(*other*)

Element wise right multiplication (*) for ttensors (only scalars supported)

Parameters
other (*float, int*)

Returns
pyttb.ttensor

ttv(*vector*, *dims=None*, *exclude_dims=None*)

TTensor times vector

Parameters

- **vector** (Numpy.ndarray, list[Numpy.ndarray])
- **dims** (Numpy.ndarray, int)

mttkrp(*U*, *n*)

Matricized tensor times Khatri-Rao product for ttensors.

Parameters

- **U** (*array of matrices or ktensor*)
- **n** (*multiplies by all modes except n*)

Returns
numpy.ndarray

norm()

Compute the norm of a ttensor. :returns: **norm** (*float, Frobenius norm of Tensor*)

permute(*order*)

Permute dimensions for a ttensor

Parameters
order (Numpy.ndarray)

Returns
pyttb.ttensor

ttm(*matrix*, *dims=None*, *exclude_dims=None*, *transpose=False*)

Tensor times matrix for ttensor

Parameters

- **matrix** (Numpy.ndarray, list[Numpy.ndarray])
- **dims** (Numpy.ndarray, int)
- **transpose** (*bool*)

reconstruct(*samples=None*, *modes=None*)

Reconstruct or partially reconstruct tensor from ttensor.

Parameters

- **samples** (Numpy.ndarray, list[Numpy.ndarray])
- **modes** (Numpy.ndarray, list[Numpy.ndarray])

Returns
pyttb.ttensor

nvecs(*n*, *r*, *flipsign=True*)

Compute the leading mode-n vectors for a ttensor.

Parameters

- **n** (*mode for tensor matricization*)
- **r** (*number of eigenvalues*)
- **flipsign** (*Make each column's largest element positive if true*)

Returns
numpy.ndarray

__module__ = 'pyttb.ttensor'

pyttb.tenmat

class pyttb.tenmat.**tenmat**

Bases: object

TENMAT Store tensor as a matrix.

__init__()

TENSOR Create empty tensor.

classmethod from_data(*data*, *rdims*, *cdims=None*, *tshape=None*)

classmethod from_tensor_type(*source*, *rdims=None*, *cdims=None*, *cdims_cyclic=None*)

ctranspose()

Complex conjugate transpose for tenmat.

Returns
pyttb.tenmat

double()

Convert tenmat to an array of doubles

Returns
numpy.ndarray – copy of tenmat data

end(*k*)

Last index of indexing expression for tenmat

Parameters
k (*int*) – dimension for subscripted indexing

Returns
int (*index*)

property ndims

Return the number of dimensions of a tenmat

Returns
int

norm()

Frobenius norm of a tenmat.

Returns
float

property shape

Return the shape of a tenmat

Returns
tuple

__setitem__(*key*, *value*)

SUBSASGN Subscripted assignment for a tensor.

__getitem__(*item*)

SUBSREF Subscripted reference for tenmat.

Parameters
item

Returns
numpy.ndarray, float, int

__mul__(*other*)

Multiplies two tenmat objects.

Parameters
other (*pyttb.tenmat*)

Returns
pyttb.tenmat

__rmul__(*other*)

Multiplies two tenmat objects.

Parameters
other (*pyttb.tenmat*)

Returns
pyttb.tenmat

__add__(*other*)

__radd__(*other*)

Reverse binary addition (+) for tenmats

Parameters
other (*pyttb.tenmat*, float, int)

Returns
pyttb.tenmat

__sub__(*other*)

__rsub__(*other*)

Reverse binary subtraction (-) for tenmats

Parameters
other (*pyttb.tenmat*, float, int)

Returns
pyttb.tenmat

__pos__()

Unary plus (+) for tenmats

Returns
pyttb.tenmat – copy of tenmat

__neg__()

Unary minus (-) for tenmats

Returns
pyttb.tenmat – copy of tenmat

__repr__()

String representation of a tenmat.

Returns
str – Contains the shape, row indices (rindices), column indices (cindices) and data as strings on different lines.

__str__()

String representation of a tenmat.

Returns

str – Contains the shape, row indices (rindices), column indices (cindices) and data as strings on different lines.

__module__ = 'pyttb.tenmat'

2.1.2 Algorithms

pyttb.cp_als

pyttb.cp_als.**cp_als**(*input_tensor*, *rank*, *stoptol=0.0001*, *maxiters=1000*, *dimorder=None*, *init='random'*, *printitn=1*, *fixsigns=True*)

Compute CP decomposition with alternating least squares

Parameters

- **input_tensor** (*pyttb.tensor* or *pyttb.sptensor* or *pyttb.ktensor*)
- **rank** (*int*) – Rank of the decomposition
- **stoptol** (*float*) – Tolerance used for termination - when the change in the fitness function in successive iterations drops below this value, the iterations terminate (default: 1e-4)
- **dimorder** (*list*) – Order to loop through dimensions (default: [range(tensor.ndims)])
- **maxiters** (*int*) – Maximum number of iterations (default: 1000)
- **init** (str or *pyttb.ktensor*) –

  Initial guess (default: "random")

  – "random": initialize using a *pyttb.ktensor* with values chosen from a Normal distribution with mean 0 and standard deviation 1

  – "nvecs": initialize factor matrices of a *pyttb.ktensor* using the eigenvectors of the outer product of the matricized input tensor

  – *pyttb.ktensor*: initialize using a specific *pyttb.ktensor* as input - must be the same shape as the input tensor and have the same rank as the input rank
- **printitn** (*int*) – Number of iterations to perform before printing iteration status - 0 for no status printing (default: 1)
- **fixsigns** (*bool*) – Align the signs of the columns of the factorization to align with the input tensor data (default: True)

Returns

- **M** (*pyttb.ktensor*) – Resulting ktensor from CP-ALS factorization
- **Minit** (*pyttb.ktensor*) – Initial guess
- **output** (*dict*) –

  Information about the computation. Dictionary keys:

  – *params* : tuple of (stoptol, maxiters, printitn, dimorder)

  – *iters*: number of iterations performed

  – *normresidual*: norm of the difference between the input tensor and ktensor factorization

- *fit*: value of the fitness function (fraction of tensor data explained by the model)

Example

Random initialization causes slight pertubation in intermediate results. ... is our place holder for these numeric values. Example using default values ("random" initialization):

```
>>> weights = np.array([1., 2.])
>>> fm0 = np.array([[1., 2.], [3., 4.]])
>>> fm1 = np.array([[5., 6.], [7., 8.]])
>>> K = ttb.ktensor.from_data(weights, [fm0, fm1])
>>> np.random.seed(1)
>>> M, Minit, output = ttb.cp_als(K.full(), 2)
CP_ALS:
 Iter 0: f = ... f-delta = ...
 Iter 1: f = ... f-delta = ...
 Final f = ...
>>> print(M)
ktensor of shape 2 x 2
weights=[108.4715... 8.6114...]
factor_matrices[0] =
[[0.4187... 0.3989...]
 [0.9080... 0.9169...]]
factor_matrices[1] =
[[0.6188... 0.2581...]
 [0.7854... 0.9661...]]
>>> print(Minit)
ktensor of shape 2 x 2
weights=[1. 1.]
factor_matrices[0] =
[[4.1702...e-01 7.2032...e-01]
 [1.1437...e-04 3.0233...e-01]]
factor_matrices[1] =
[[0.1467... 0.0923...]
 [0.1862... 0.3455...]]
>>> print(output)
{'params': (0.0001, 1000, 1, [0, 1]), 'iters': 1, 'normresidual': ..., 'fit': ...}
```

Example using "nvecs" initialization:

```
>>> M, Minit, output = ttb.cp_als(K.full(), 2, init="nvecs")
CP_ALS:
 Iter 0: f = ... f-delta = ...
 Iter 1: f = ... f-delta = ...
 Final f = ...
```

Example using *pyttb.ktensor* initialization:

```
>>> M, Minit, output = ttb.cp_als(K.full(), 2, init=K)
CP_ALS:
 Iter 0: f = ... f-delta = ...
 Iter 1: f = ... f-delta = ...
 Final f = ...
```

pyttb.cp_apr

pyttb.cp_apr.**cp_apr**(*input_tensor*, *rank*, *algorithm='mu'*, *stoptol=0.0001*, *stoptime=1000000.0*, *maxiters=1000*, *init='random'*, *maxinneriters=10*, *epsDivZero=1e-10*, *printitn=1*, *printinneritn=0*, *kappa=0.01*, *kappatol=1e-10*, *epsActive=1e-08*, *mu0=1e-05*, *precompinds=True*, *inexact=True*, *lbfgsMem=3*)

Compute non-negative CP with alternating Poisson regression.

Parameters

- **input_tensor** (*pyttb.tensor* or *pyttb.sptensor*)
- **rank** (*int*) – Rank of the decomposition
- **algorithm** (*str*) – in {'mu', 'pdnr, 'pqnr'}
- **stoptol** (*float*) – Tolerance on overall KKT violation
- **stoptime** (*float*) – Maximum number of seconds to run
- **maxiters** (*int*) – Maximum number of iterations
- **init** (str or *pyttb.ktensor*) – Initial guess
- **maxinneriters** (*int*) – Maximum inner iterations per outer iteration
- **epsDivZero** (*float*) – Safeguard against divide by zero
- **printitn** (*int*) – Print every n outer iterations, 0 for none
- **printinneritn** (*int*) – Print every n inner iterations
- **kappa** (*int*) – MU ALGORITHM PARAMETER: Offset to fix complementary slackness
- **kappatol** – MU ALGORITHM PARAMETER: Tolerance on complementary slackness
- **epsActive** (*float*) – PDNR & PQNR ALGORITHM PARAMETER: Bertsekas tolerance for active set
- **mu0** (*float*) – PDNR ALGORITHM PARAMETER: Initial Damping Parameter
- **precompinds** (*bool*) – PDNR & PQNR ALGORITHM PARAMETER: Precompute sparse tensor indices
- **inexact** (*bool*) – PDNR ALGORITHM PARAMETER: Compute inexact Newton steps
- **lbfgsMem** (*int*) – PQNR ALGORITHM PARAMETER: Precompute sparse tensor indices

Returns

- **M** (*pyttb.ktensor*) – Resulting ktensor from CP APR
- **Minit** (*pyttb.ktensor*) – Initial Guess
- **output** (*dict*) – Additional output #TODO document this more appropriately

pyttb.cp_apr.**tt_cp_apr_mu**(*input_tensor*, *rank*, *init*, *stoptol*, *stoptime*, *maxiters*, *maxinneriters*, *epsDivZero*, *printitn*, *printinneritn*, *kappa*, *kappatol*)

Compute nonnegative CP with alternating Poisson regression.

Parameters

- **input_tensor** (*pyttb.tensor* or *pyttb.sptensor*)
- **rank** (*int*) – Rank of the decomposition
- **init** (*pyttb.ktensor*) – Initial guess

- **stoptol** (*float*) – Tolerance on overall KKT violation
- **stoptime** (*float*) – Maximum number of seconds to run
- **maxiters** (*int*) – Maximum number of iterations
- **maxinneriters** (*int*) – Maximum inner iterations per outer iteration
- **epsDivZero** (*float*) – Safeguard against divide by zero
- **printitn** (*int*) – Print every n outer iterations, 0 for none
- **printinneritn** (*int*) – Print every n inner iterations
- **kappa** (*int*) – MU ALGORITHM PARAMETER: Offset to fix complementary slackness
- **kappatol** – MU ALGORITHM PARAMETER: Tolerance on complementary slackness

Notes

REFERENCE: E. C. Chi and T. G. Kolda. On Tensors, Sparsity, and Nonnegative Factorizations, arXiv:1112.2414 [math.NA], December 2011, URL: http://arxiv.org/abs/1112.2414. Submitted for publication.

pyttb.cp_apr.**tt_cp_apr_pdnr**(*input_tensor*, *rank*, *init*, *stoptol*, *stoptime*, *maxiters*, *maxinneriters*, *epsDivZero*, *printitn*, *printinneritn*, *epsActive*, *mu0*, *precompinds*, *inexact*)

Compute nonnegative CP with alternating Poisson regression computes an estimate of the best rank-R CP model of a tensor X using an alternating Poisson regression. The algorithm solves "row subproblems" in each alternating subproblem, using a Hessian of size R^2.

Parameters

- **# TODO it looks like this method of define union helps the typ hinting better than or**
- **input_tensor** (Union[pyttb.tensor,:class:*pyttb.sptensor*])
- **rank** (*int*) – Rank of the decomposition
- **init** (str or pyttb.ktensor) – Initial guess
- **stoptol** (*float*) – Tolerance on overall KKT violation
- **stoptime** (*float*) – Maximum number of seconds to run
- **maxiters** (*int*) – Maximum number of iterations
- **maxinneriters** (*int*) – Maximum inner iterations per outer iteration
- **epsDivZero** (*float*) – Safeguard against divide by zero
- **printitn** (*int*) – Print every n outer iterations, 0 for none
- **printinneritn** (*int*) – Print every n inner iterations
- **epsActive** (*float*) – PDNR & PQNR ALGORITHM PARAMETER: Bertsekas tolerance for active set
- **mu0** (*float*) – PDNR ALGORITHM PARAMETER: Initial Damping Parameter
- **precompinds** (*bool*) – PDNR & PQNR ALGORITHM PARAMETER: Precompute sparse tensor indices
- **inexact** (*bool*) – PDNR ALGORITHM PARAMETER: Compute inexact Newton steps

Returns

 # TODO detail return dictionary

Notes

REFERENCE: Samantha Hansen, Todd Plantenga, Tamara G. Kolda. Newton-Based Optimization for Nonnegative Tensor Factorizations, arXiv:1304.4964 [math.NA], April 2013, URL: http://arxiv.org/abs/1304.4964. Submitted for publication.

pyttb.cp_apr.**tt_cp_apr_pqnr**(*input_tensor*, *rank*, *init*, *stoptol*, *stoptime*, *maxiters*, *maxinneriters*, *epsDivZero*, *printitn*, *printinneritn*, *epsActive*, *lbfgsMem*, *precompinds*)

Compute nonnegative CP with alternating Poisson regression.

tt_cp_apr_pdnr computes an estimate of the best rank-R CP model of a tensor X using an alternating Poisson regression. The algorithm solves "row subproblems" in each alternating subproblem, using a Hessian of size R^2. The function is typically called by cp_apr.

The model is solved by nonlinear optimization, and the code literally minimizes the negative of log-likelihood. However, printouts to the console reverse the sign to show maximization of log-likelihood.

mu0: float
PDNR ALGORITHM PARAMETER: Initial Damping Parameter

precompinds: bool
PDNR & PQNR ALGORITHM PARAMETER: Precompute sparse tensor indices

inexact: bool
PDNR ALGORITHM PARAMETER: Compute inexact Newton steps

Parameters

- **input_tensor** (Union[pyttb.tensor,:class:*pyttb.sptensor*])
- **rank** (*int*) – Rank of the decomposition
- **init** (str or pyttb.ktensor) – Initial guess
- **stoptol** (*float*) – Tolerance on overall KKT violation
- **stoptime** (*float*) – Maximum number of seconds to run
- **maxiters** (*int*) – Maximum number of iterations
- **maxinneriters** (*int*) – Maximum inner iterations per outer iteration
- **epsDivZero** (*float*) – Safeguard against divide by zero
- **printitn** (*int*) – Print every n outer iterations, 0 for none
- **printinneritn** (*int*) – Print every n inner iterations
- **epsActive** (*float*) – PDNR & PQNR ALGORITHM PARAMETER: Bertsekas tolerance for active set
- **lbfgsMem** (*int*) – Number of vector pairs to store for L-BFGS
- **precompinds**

Returns
 # TODO detail return dictionary

Notes

REFERENCE: Samantha Hansen, Todd Plantenga, Tamara G. Kolda. Newton-Based Optimization for Non-negative Tensor Factorizations, arXiv:1304.4964 [math.NA], April 2013, URL: http://arxiv.org/abs/1304.4964. Submitted for publication.

pyttb.cp_apr.**tt_calcpi_prowsubprob**(*Data*, *Model*, *rank*, *factorIndex*, *ndims*, *isSparse=False*, *sparse_indices=None*)

Compute Pi for a row subproblem.

Parameters

- **Data :class:`pyttb.sptensor` or :class:`pyttb.tensor`**
- **isSparse** (*bool*)
- **Model :class:`pyttb.ktensor`**
- **rank** (*int*)
- **factorIndex** (*int*)
- **ndims** (*int*)
- **sparse_indices** (*list*) – Indices of row subproblem nonzero elements

Returns

Pi (numpy.ndarray)

See also:

pyttb.calculatePi

pyttb.cp_apr.**calc_partials**(*isSparse*, *Pi*, *epsilon*, *data_row*, *model_row*)

Compute derivative quantities for a PDNR row subproblem.

Parameters

- **isSparse** (*bool*)
- **Pi** (numpy.ndarray)
- **epsilon** (*float*) – Prevent division by zero
- **data_row** (numpy.ndarray)
- **model_row** (numpy.ndarray)

Returns

- **phi_row** (numpy.ndarray) – gradient of row subproblem, except for a constant

  $phi\_row[r] = \sum_{j=1}^{J_n} \frac{x_j \pi_{rj}}{\sum_i^R b_i \pi_{ij}}$
- **ups_row** (numpy.ndarray) – intermediate quantity (upsilon) used for second derivatives

  $ups\_row[j] = \frac{x_j}{\left(\sum_i^R b_i \pi_{ij}\right)^2}$

pyttb.cp_apr.**getSearchDirPdnr**(*Pi*, *ups_row*, *rank*, *gradModel*, *model_row*, *mu*, *epsActSet*)

Compute the search direction for PDNR using a two-metric projection with damped Hessian

Parameters

- **Pi** (numpy.ndarray)
- **ups_row** (numpy.ndarray) – intermediate quantity (upsilon) used for second derivatives

- **rank** (*int*) – number of variables for the row subproblem
- **gradModel** (numpy.ndarray) – gradient vector for the row subproblem
- **model_row** (numpy.ndarray) – vector of variables for the row subproblem
- **mu** (*float*) – damping parameter
- **epsActSet** (*float*) – Bertsekas tolerance for active set determination

Returns

- **search_dir** (numpy.ndarray) – search direction vector
- **pred_red** (numpy.ndarray) – predicted reduction in quadratic model

pyttb.cp_apr.**tt_linesearch_prowsubprob**(*direction*, *grad*, *model_old*, *step_len*, *step_red*, *max_steps*, *suff_decr*, *isSparse*, *data_row*, *Pi*, *phi_row*, *display_warning*)

Perform a line search on a row subproblem

Parameters

- **direction** (numpy.ndarray) – search direction
- **grad** (numpy.ndarray) – gradient vector a model_old
- **model_old** (numpy.ndarray) – current variable values
- **step_len** (*float*) – initial step length, which is the maximum possible step length
- **step_red** (*float*) – step reduction factor (suggest 1/2)
- **max_steps** (*int*) – maximum number of steps to try (suggest 10)
- **suff_decr** (*float*) – sufficent decrease for convergence (suggest 1.0e-4)
- **isSparse** (*bool*) – sparsity flag for computing the objective
- **data_row** (numpy.ndarray) – row subproblem data, for computing the objective
- **Pi** (numpy.ndarray) – Pi matrix, for computing the objective
- **phi_row** (numpy.ndarray) – 1-grad, more accurate if failing over to multiplicative update
- **display_warning** (*bool*) – Flag to display warning messages or not

Returns

- **m_new** (numpy.ndarray) – new (improved) model values
- **num_evals** (*int*) – number of times objective was evaluated
- **f_old** (*float*) – objective value at model_old
- **f_1** (*float*) – objective value at model_old + step_len*direction
- **f_new** (*float*) – objective value at model_new

pyttb.cp_apr.**getHessian**(*upsilon*, *Pi*, *free_indices*)

Return the Hessian for one PDNR row subproblem of Model[n], for just the rows and columns corresponding to the free variables

Parameters

- **upsilon** (numpy.ndarray) – intermediate quantity (upsilon) used for second derivatives
- **Pi** (numpy.ndarray)
- **free_indices** (*list*)

Returns
Hessian (numpy.ndarray) – Sub-block of full Hessian identified by free-indices

pyttb.cp_apr.**tt_loglikelihood_row**(*isSparse*, *data_row*, *model_row*, *Pi*)

Compute log-likelihood of one row subproblem

Parameters

- **isSparse** (*bool*) – Sparsity flag
- **data_row** (numpy.ndarray) – vector of data values
- **model_row** (numpy.ndarray) – vector of model values
- **Pi** (numpy.ndarray)

Notes

The row subproblem for a given mode includes one row of matricized tensor data (x) and one row of the model (m) in the same matricized mode. Then (dense case) m: R-length vector x: J-length vector Pi: R x J matrix (sparse case) m: R-length vector x: p-length vector, where p = nnz in row of matricized data tensor Pi: R x p matrix F = - (sum_r m_r - sum_j x_j * log (m * Pi_j) where Pi_j denotes the j^th column of Pi NOTE: Rows of Pi' must sum to one

Returns
loglikelihood (*float*) – See notes for description

pyttb.cp_apr.**getSearchDirPqnr**(*model_row*, *gradModel*, *epsActSet*, *delta_model*, *delta_grad*, *rho*, *lbfgs_pos*, *iters*, *disp_warn*)

Compute the search direction by projecting with L-BFGS.

Parameters

- **model_row** (numpy.ndarray) – current variable values
- **gradModel** (numpy.ndarray) – gradient at model_row
- **epsActSet** (*float*) – Bertsekas tolerance for active set determination
- **delta_model** (numpy.ndarray) – L-BFGS array of variable deltas
- **delta_grad** (numpy.ndarray) – L-BFGS array of gradient deltas
- **rho**
- **lbfgs_pos** (*int*) – pointer into L-BFGS arrays
- **iters**
- **disp_warn** (*bool*)

Returns
direction (numpy.ndarray) – Search direction based on current L-BFGS and grad

Notes

Adapted from MATLAB code of Dongmin Kim and Suvrit Sra written in 2008. Modified extensively to solve row subproblems and use a better linesearch; for details see REFERENCE: Samantha Hansen, Todd Plantenga, Tamara G. Kolda. Newton-Based Optimization for Nonnegative Tensor Factorizations, arXiv:1304.4964 [math.NA], April 2013, URL: http://arxiv.org/abs/1304.4964. Submitted for publication.

pyttb.cp_apr.**calc_grad**(*isSparse*, *Pi*, *eps_div_zero*, *data_row*, *model_row*)

Compute the gradient for a PQNR row subproblem

Parameters

- **isSparse** (*bool*)
- **Pi** (numpy.ndarray)
- **eps_div_zero** (*float*)
- **data_row** (numpy.ndarray)
- **model_row** (numpy.ndarray)

Returns

- **phi_row** (numpy.ndarray)
- **grad_row** (numpy.ndarray)

pyttb.cp_apr.**calculatePi**(*Data*, *Model*, *rank*, *factorIndex*, *ndims*)

Helper function to calculate Pi matrix # TODO verify what pi is

Parameters

- **Data** (*pyttb.sptensor* or *pyttb.tensor*)
- **Model** (*pyttb.ktensor*)
- **rank** (*int*)
- **factorIndex** (*int*)
- **ndims** (*int*)

Returns

Pi (numpy.ndarray)

pyttb.cp_apr.**calculatePhi**(*Data*, *Model*, *rank*, *factorIndex*, *Pi*, *epsilon*)

Parameters

- **Data** (*pyttb.sptensor* or *pyttb.tensor*)
- **Model** (*pyttb.ktensor*)
- **rank** (*int*)
- **factorIndex** (*int*)
- **Pi** (numpy.ndarray)
- **epsilon** (*float*)

pyttb.cp_apr.**tt_loglikelihood**(*Data*, *Model*)

Compute log-likelihood of data with model.

Parameters

- **Data** (*pyttb.sptensor* or *pyttb.tensor*)
- **Model** (*pyttb.ktensor*)

Returns

loglikelihood (*float*) –

- (sum_i m_i - x_i * log_i) where i is a multiindex across all tensor dimensions.

Notes

We define for any x 0*log(x)=0, such that if our true data is 0 the loglikelihood is the value of the model.

pyttb.cp_apr.**vectorizeForMu**(*matrix*)

Helper Function to unravel matrix into vector

Parameters

matrix (*numpy.ndarray*)

Returns

matrix (*numpy.ndarray*) – len(matrix.shape)==1

pyttb.hosvd

Higher Order SVD Implementation

pyttb.hosvd.**hosvd**(*input_tensor*, *tol: float*, *verbosity: float = 1*, *dimorder: Optional[List[int]] = None*, *sequential: bool = True*, *ranks: Optional[List[int]] = None*)

Compute sequentially-truncated higher-order SVD (Tucker).

Computes a Tucker decomposition with relative error specified by tol, i.e., it computes a ttensor T such that ||X-T||/||X|| <= tol.

Parameters

- **input_tensor** (*Tensor to factor*)
- **tol** (*Relative error to stop at*)
- **verbosity** (*Print level*)
- **dimorder** (*Order to loop through dimensions*)
- **sequential** (*Use sequentially-truncated version*)
- **ranks** (*Specify ranks to consider rather than computing*)

Example

```
>>> data = np.array([[29, 39.], [63., 85.]])
>>> tol = 1e-4
>>> disable_printing = -1
>>> tensorInstance = ttb.tensor().from_data(data)
>>> result = hosvd(tensorInstance, tol, verbosity=disable_printing)
>>> ((result.full() - tensorInstance).norm() / tensorInstance.norm()) < tol
True
```

pyttb.tucker_als

pyttb.tucker_als.**tucker_als**(*input_tensor*, *rank*, *stoptol=0.0001*, *maxiters=1000*, *dimorder=None*, *init='random'*, *printitn=1*)

Compute Tucker decomposition with alternating least squares

Parameters

- **input_tensor** (*pyttb.tensor*)
- **rank** (*int, list[int]*) – Rank of the decomposition(s)
- **stoptol** (*float*) – Tolerance used for termination - when the change in the fitness function in successive iterations drops below this value, the iterations terminate (default: 1e-4)
- **dimorder** (*list*) – Order to loop through dimensions (default: [range(tensor.ndims)])
- **maxiters** (*int*) – Maximum number of iterations (default: 1000)
- **init** (*str or list[np.ndarray]*) –

  Initial guess (default: "random")

  – "random": initialize using a *pyttb.ttensor* with values chosen from a Normal distribution with mean 1 and standard deviation 0

  – "nvecs": initialize factor matrices of a *pyttb.ttensor* using the eigenvectors of the outer product of the matricized input tensor

  – *pyttb.ttensor*: initialize using a specific *pyttb.ttensor* as input - must be the same shape as the input tensor and have the same rank as the input rank
- **printitn** (*int*) – Number of iterations to perform before printing iteration status - 0 for no status printing (default: 1)

Returns

- **M** (*pyttb.ttensor*) – Resulting ttensor from Tucker-ALS factorization
- **Minit** (*pyttb.ttensor*) – Initial guess
- **output** (*dict*) –

  Information about the computation. Dictionary keys:

  – *params* : tuple of (stoptol, maxiters, printitn, dimorder)

  – *iters*: number of iterations performed

  – *normresidual*: norm of the difference between the input tensor and ktensor factorization

  – *fit*: value of the fitness function (fraction of tensor data explained by the model)

CHAPTER

THREE

HOW TO CITE

Please see references for how to cite a variety of algorithms implemented in this project.

3.1 BibTex Entries: Tensor Toolbox for Python

```
@misc{TTB_Software,
 author = {Brett W. Bader and Tamara G. Kolda and others},
 title = {MATLAB Tensor Toolbox Version 3.0-dev},
 howpublished = {Available online},
 month = oct,
 year = {2017},
 url = {https://www.tensortoolbox.org}
}
@article{TTB_Dense,
  author = {Brett W. Bader and Tamara G. Kolda},
  title = {Algorithm 862: {MATLAB} tensor classes for fast algorithm prototyping},
  journal = {ACM Transactions on Mathematical Software},
  month = dec,
  year = {2006},
  volume = {32},
  number = {4},
  pages = {635-653},
  doi = {10.1145/1186785.1186794}
}
@article{TTB_Sparse,
  author = {Brett W. Bader and Tamara G. Kolda},
  title = {Efficient {MATLAB} computations with sparse and factored tensors},
  journal = {SIAM Journal on Scientific Computing},
  month = dec,
  year = {2007},
  volume = {30},
  number = {1},
  pages = {205-231},
  doi = {10.1137/060676489}
}
@article{TTB_CPOPT,
  author = {Evrim Acar and Daniel M. Dunlavy and Tamara G. Kolda},
  title = {A Scalable Optimization Approach for Fitting Canonical Tensor Decompositions},
  journal = {Journal of Chemometrics},
```

(continues on next page)

(continued from previous page)

```
  month = feb,
  year = {2011},
  volume = {25},
  number = {2},
  pages = {67-86},
  doi = {10.1002/cem.1335}
}
@article{TTB_CPWOPT,
  author = {Evrim Acar and Daniel M. Dunlavy and Tamara G. Kolda and Morten M{\o}rup},
  title = {Scalable Tensor Factorizations for Incomplete Data},
  journal = {Chemometrics and Intelligent Laboratory Systems},
  month = mar,
  year = {2011},
  volume = {106},
  number = {1},
  pages = {41-56},
  doi = {10.1016/j.chemolab.2010.08.004}
}
@article{TTB_SSHOPM,
  author = {Tamara G. Kolda and Jackson R. Mayo},
  title = {Shifted Power Method for Computing Tensor Eigenpairs},
  journal = {SIAM Journal on Matrix Analysis and Applications},
  month = oct,
  year = {2011},
  volume = {32},
  number = {4},
  pages = {1095-1124},
  doi = {10.1137/100801482}
}
@Article{TTB_EIGGEAP,
  title = {An Adaptive Shifted Power Method for Computing Generalized Tensor Eigenpairs},
  author = {Tamara G. Kolda and Jackson R. Mayo},
  doi = {10.1137/140951758},
  journal = {SIAM Journal on Matrix Analysis and Applications},
  number = {4},
  volume = {35},
  year = {2014},
  month = dec,
  pages = {1563-1581},
}
@Article{TTB_CPAPR,
  title = {On Tensors, Sparsity, and Nonnegative Factorizations},
  author = {Eric C. Chi and Tamara G. Kolda},
  doi = {10.1137/110859063},
  journal = {SIAM Journal on Matrix Analysis and Applications},
  number = {4},
  volume = {33},
  year = {2012},
  month = dec,
  pages = {1272-1299},
}
@Article{TTB_CPAPRB,
```

(continues on next page)

(continued from previous page)

```
  author = {Samantha Hansen and Todd Plantenga and Tamara G. Kolda},
  title = {Newton-Based Optimization for {Kullback-Leibler} Nonnegative Tensor␣
↪Factorizations},
  journal = {Optimization Methods and Software},
  volume = {30},
  number = {5},
  pages = {1002-1029},
  month = {April},
  year = {2015},
  doi = {10.1080/10556788.2015.1009977},
}
@article{TTB_CPSYM,
  author = {Tamara G. Kolda},
  title = {Numerical Optimization for Symmetric Tensor Decomposition},
  journal = {Mathematical Programming B},
  volume = {151},
  number = {1},
  pages = {225-248},
  month = apr,
  year = {2015},
  doi = {10.1007/s10107-015-0895-0},
}
@misc{TTB_CPRALS,
  author = {Casey Battaglino and Grey Ballard and Tamara G. Kolda},
  title = {A Practical Randomized {CP} Tensor Decomposition},
  howpublished = {arXiv:1701.06600},
  month = jan,
  year = {2017},
  eprint = {1701.06600},
  eprintclass = {cs.NA},
}
@inproceedings{TTB_MET,
  author = {Tamara G. Kolda and Jimeng Sun},
  title = {Scalable Tensor Decompositions for Multi-aspect Data Mining},
  booktitle = {ICDM 2008: Proceedings of the 8th IEEE International Conference on Data␣
↪Mining},
  month = dec,
  year = {2008},
  pages = {363-372},
  doi = {10.1109/ICDM.2008.89}
}
```

CHAPTER

FOUR

CONTACT

Please email dmdunla@sandia.gov with any questions about pyttb that cannot be resolved via issue reporting. Stories of its usefulness are especially welcome. We will try to respond to every email may not always be successful due to the volume of emails.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

p

pyttb.cp_als, 62
pyttb.cp_apr, 64
pyttb.hosvd, 71
pyttb.sptensor, 32
pyttb.tenmat, 60
pyttb.tensor, 43
pyttb.ttensor, 56
pyttb.tucker_als, 72

INDEX

Symbols

__add__() (*pyttb.ktensor method*), 29
__add__() (*pyttb.sptensor.sptensor method*), 40
__add__() (*pyttb.tenmat.tenmat method*), 61
__add__() (*pyttb.tensor.tensor method*), 53
__annotations__ (*pyttb.tensor.tensor attribute*), 55
__eq__() (*pyttb.sptensor.sptensor method*), 40
__eq__() (*pyttb.tensor.tensor method*), 52
__ge__() (*pyttb.sptensor.sptensor method*), 41
__ge__() (*pyttb.tensor.tensor method*), 53
__getitem__() (*pyttb.ktensor method*), 29
__getitem__() (*pyttb.sptensor.sptensor method*), 38
__getitem__() (*pyttb.tenmat.tenmat method*), 60
__getitem__() (*pyttb.tensor.tensor method*), 52
__gt__() (*pyttb.sptensor.sptensor method*), 41
__gt__() (*pyttb.tensor.tensor method*), 53
__hash__ (*pyttb.sptensor.sptensor attribute*), 41
__hash__ (*pyttb.tensor.tensor attribute*), 55
__init__() (*pyttb.sptensor.sptensor method*), 32
__init__() (*pyttb.tenmat.tenmat method*), 60
__init__() (*pyttb.tensor.tensor method*), 43
__init__() (*pyttb.ttensor.ttensor method*), 56
__le__() (*pyttb.sptensor.sptensor method*), 41
__le__() (*pyttb.tensor.tensor method*), 53
__lt__() (*pyttb.sptensor.sptensor method*), 41
__lt__() (*pyttb.tensor.tensor method*), 53
__module__ (*pyttb.ktensor attribute*), 32
__module__ (*pyttb.sptensor.sptensor attribute*), 42
__module__ (*pyttb.tenmat.tenmat attribute*), 62
__module__ (*pyttb.tensor.tensor attribute*), 55
__module__ (*pyttb.ttensor.ttensor attribute*), 59
__mul__() (*pyttb.ktensor method*), 31
__mul__() (*pyttb.sptensor.sptensor method*), 40
__mul__() (*pyttb.tenmat.tenmat method*), 61
__mul__() (*pyttb.tensor.tensor method*), 54
__mul__() (*pyttb.ttensor.ttensor method*), 58
__ne__() (*pyttb.sptensor.sptensor method*), 40
__ne__() (*pyttb.tensor.tensor method*), 52
__neg__() (*pyttb.ktensor method*), 30
__neg__() (*pyttb.sptensor.sptensor method*), 40
__neg__() (*pyttb.tenmat.tenmat method*), 61
__neg__() (*pyttb.tensor.tensor method*), 54
__neg__() (*pyttb.ttensor.ttensor method*), 58
__pos__() (*pyttb.ktensor method*), 30
__pos__() (*pyttb.sptensor.sptensor method*), 40
__pos__() (*pyttb.tenmat.tenmat method*), 61
__pos__() (*pyttb.tensor.tensor method*), 54
__pos__() (*pyttb.ttensor.ttensor method*), 58
__pow__() (*pyttb.tensor.tensor method*), 53
__radd__() (*pyttb.tenmat.tenmat method*), 61
__radd__() (*pyttb.tensor.tensor method*), 53
__repr__() (*pyttb.ktensor method*), 31
__repr__() (*pyttb.sptensor.sptensor method*), 41
__repr__() (*pyttb.tenmat.tenmat method*), 61
__repr__() (*pyttb.tensor.tensor method*), 54
__repr__() (*pyttb.ttensor.ttensor method*), 57
__rmul__() (*pyttb.ktensor method*), 31
__rmul__() (*pyttb.sptensor.sptensor method*), 40
__rmul__() (*pyttb.tenmat.tenmat method*), 61
__rmul__() (*pyttb.tensor.tensor method*), 54
__rmul__() (*pyttb.ttensor.ttensor method*), 58
__rsub__() (*pyttb.tenmat.tenmat method*), 61
__rtruediv__() (*pyttb.sptensor.sptensor method*), 41
__rtruediv__() (*pyttb.tensor.tensor method*), 54
__setitem__() (*pyttb.ktensor method*), 30
__setitem__() (*pyttb.sptensor.sptensor method*), 39
__setitem__() (*pyttb.tenmat.tenmat method*), 60
__setitem__() (*pyttb.tensor.tensor method*), 52
__str__() (*pyttb.ktensor method*), 31
__str__() (*pyttb.sptensor.sptensor method*), 42
__str__() (*pyttb.tenmat.tenmat method*), 61
__str__() (*pyttb.tensor.tensor method*), 54
__str__() (*pyttb.ttensor.ttensor method*), 57
__sub__() (*pyttb.ktensor method*), 31
__sub__() (*pyttb.sptensor.sptensor method*), 40
__sub__() (*pyttb.tenmat.tenmat method*), 61
__sub__() (*pyttb.tensor.tensor method*), 53
__truediv__() (*pyttb.sptensor.sptensor method*), 41
__truediv__() (*pyttb.tensor.tensor method*), 54

A

allsubs() (*pyttb.sptensor.sptensor method*), 34
arrange() (*pyttb.ktensor method*), 10

C

calc_grad() (*in module pyttb.cp_apr*), 70
calc_partials() (*in module pyttb.cp_apr*), 67
calculatePhi() (*in module pyttb.cp_apr*), 70
calculatePi() (*in module pyttb.cp_apr*), 70
collapse() (*pyttb.sptensor.sptensor method*), 34
collapse() (*pyttb.tensor.tensor method*), 44
contract() (*pyttb.sptensor.sptensor method*), 34
contract() (*pyttb.tensor.tensor method*), 44
copy() (*pyttb.ktensor method*), 12
cp_als() (*in module pyttb.cp_als*), 62
cp_apr() (*in module pyttb.cp_apr*), 64
ctranspose() (*pyttb.tenmat.tenmat method*), 60

D

data (*pyttb.tensor.tensor attribute*), 43
double() (*pyttb.ktensor method*), 13
double() (*pyttb.sptensor.sptensor method*), 35
double() (*pyttb.tenmat.tenmat method*), 60
double() (*pyttb.tensor.tensor method*), 45
double() (*pyttb.ttensor.ttensor method*), 57

E

elemfun() (*pyttb.sptensor.sptensor method*), 35
end() (*pyttb.ktensor method*), 14
end() (*pyttb.sptensor.sptensor method*), 35
end() (*pyttb.tenmat.tenmat method*), 60
end() (*pyttb.tensor.tensor method*), 45
exp() (*pyttb.tensor.tensor method*), 45
extract() (*pyttb.ktensor method*), 14
extract() (*pyttb.sptensor.sptensor method*), 35

F

find() (*pyttb.sptensor.sptensor method*), 35
find() (*pyttb.tensor.tensor method*), 45
fixsigns() (*pyttb.ktensor method*), 15
from_aggregator() (*pyttb.sptensor.sptensor class method*), 33
from_data() (*pyttb.ktensor class method*), 5
from_data() (*pyttb.sptensor.sptensor class method*), 32
from_data() (*pyttb.tenmat.tenmat class method*), 60
from_data() (*pyttb.tensor.tensor class method*), 43
from_data() (*pyttb.ttensor.ttensor class method*), 56
from_factor_matrices() (*pyttb.ktensor class method*), 7
from_function() (*pyttb.ktensor class method*), 8
from_function() (*pyttb.sptensor.sptensor class method*), 33
from_function() (*pyttb.tensor.tensor class method*), 44
from_tensor_type() (*pyttb.ktensor class method*), 6
from_tensor_type() (*pyttb.sptensor.sptensor class method*), 33
from_tensor_type() (*pyttb.tenmat.tenmat class method*), 60
from_tensor_type() (*pyttb.tensor.tensor class method*), 43
from_tensor_type() (*pyttb.ttensor.ttensor class method*), 57
from_vector() (*pyttb.ktensor class method*), 9
full() (*pyttb.ktensor method*), 16
full() (*pyttb.sptensor.sptensor method*), 35
full() (*pyttb.tensor.tensor method*), 46
full() (*pyttb.ttensor.ttensor method*), 57

G

getHessian() (*in module pyttb.cp_apr*), 68
getSearchDirPdnr() (*in module pyttb.cp_apr*), 67
getSearchDirPqnr() (*in module pyttb.cp_apr*), 69

H

hosvd() (*in module pyttb.hosvd*), 71

I

innerprod() (*pyttb.ktensor method*), 16
innerprod() (*pyttb.sptensor.sptensor method*), 35
innerprod() (*pyttb.tensor.tensor method*), 46
innerprod() (*pyttb.ttensor.ttensor method*), 58
isequal() (*pyttb.ktensor method*), 17
isequal() (*pyttb.sptensor.sptensor method*), 36
isequal() (*pyttb.tensor.tensor method*), 46
isequal() (*pyttb.ttensor.ttensor method*), 57
issymmetric() (*pyttb.ktensor method*), 17
issymmetric() (*pyttb.tensor.tensor method*), 46

K

ktensor (*class in pyttb*), 5

L

logical_and() (*pyttb.sptensor.sptensor method*), 36
logical_and() (*pyttb.tensor.tensor method*), 47
logical_not() (*pyttb.sptensor.sptensor method*), 36
logical_not() (*pyttb.tensor.tensor method*), 47
logical_or() (*pyttb.sptensor.sptensor method*), 36
logical_or() (*pyttb.tensor.tensor method*), 47
logical_xor() (*pyttb.sptensor.sptensor method*), 36
logical_xor() (*pyttb.tensor.tensor method*), 48

M

mask() (*pyttb.ktensor method*), 18
mask() (*pyttb.sptensor.sptensor method*), 36
mask() (*pyttb.tensor.tensor method*), 48
module
    pyttb.cp_als, 62
    pyttb.cp_apr, 64
    pyttb.hosvd, 71

pyttb.sptensor, 32
pyttb.tenmat, 60
pyttb.tensor, 43
pyttb.ttensor, 56
pyttb.tucker_als, 72
mttkrp() (*pyttb.ktensor method*), 18
mttkrp() (*pyttb.sptensor.sptensor method*), 36
mttkrp() (*pyttb.tensor.tensor method*), 48
mttkrp() (*pyttb.ttensor.ttensor method*), 58

N

ncomponents (*pyttb.ktensor property*), 19
ndims (*pyttb.ktensor property*), 19
ndims (*pyttb.sptensor.sptensor property*), 37
ndims (*pyttb.tenmat.tenmat property*), 60
ndims (*pyttb.tensor.tensor property*), 49
ndims (*pyttb.ttensor.ttensor property*), 57
nnz (*pyttb.sptensor.sptensor property*), 37
nnz (*pyttb.tensor.tensor property*), 49
norm() (*pyttb.ktensor method*), 19
norm() (*pyttb.sptensor.sptensor method*), 37
norm() (*pyttb.tenmat.tenmat method*), 60
norm() (*pyttb.tensor.tensor method*), 49
norm() (*pyttb.ttensor.ttensor method*), 59
normalize() (*pyttb.ktensor method*), 20
nvecs() (*pyttb.ktensor method*), 20
nvecs() (*pyttb.sptensor.sptensor method*), 37
nvecs() (*pyttb.tensor.tensor method*), 49
nvecs() (*pyttb.ttensor.ttensor method*), 59

O

ones() (*pyttb.sptensor.sptensor method*), 37

P

permute() (*pyttb.ktensor method*), 21
permute() (*pyttb.sptensor.sptensor method*), 37
permute() (*pyttb.tensor.tensor method*), 50
permute() (*pyttb.ttensor.ttensor method*), 59
pyttb.cp_als
module, 62
pyttb.cp_apr
module, 64
pyttb.hosvd
module, 71
pyttb.sptensor
module, 32
pyttb.tenmat
module, 60
pyttb.tensor
module, 43
pyttb.ttensor
module, 56
pyttb.tucker_als
module, 72

R

reconstruct() (*pyttb.ttensor.ttensor method*), 59
redistribute() (*pyttb.ktensor method*), 22
reshape() (*pyttb.sptensor.sptensor method*), 37
reshape() (*pyttb.tensor.tensor method*), 50

S

scale() (*pyttb.sptensor.sptensor method*), 37
score() (*pyttb.ktensor method*), 23
shape (*pyttb.ktensor property*), 22
shape (*pyttb.tenmat.tenmat property*), 60
shape (*pyttb.tensor.tensor attribute*), 43
shape (*pyttb.ttensor.ttensor property*), 57
spmatrix() (*pyttb.sptensor.sptensor method*), 38
sptendiag() (*in module pyttb.sptensor*), 42
sptenrand() (*in module pyttb.sptensor*), 42
sptensor (*class in pyttb.sptensor*), 32
squeeze() (*pyttb.sptensor.sptensor method*), 38
squeeze() (*pyttb.tensor.tensor method*), 50
subdims() (*pyttb.sptensor.sptensor method*), 38
symmetrize() (*pyttb.ktensor method*), 24
symmetrize() (*pyttb.tensor.tensor method*), 51

T

tendiag() (*in module pyttb.tensor*), 56
tenmat (*class in pyttb.tenmat*), 60
tenones() (*in module pyttb.tensor*), 55
tenrand() (*in module pyttb.tensor*), 55
tensor (*class in pyttb.tensor*), 43
tenzeros() (*in module pyttb.tensor*), 55
tolist() (*pyttb.ktensor method*), 24
tovec() (*pyttb.ktensor method*), 25
tt_calcpi_prowsubprob() (*in module pyttb.cp_apr*), 67
tt_cp_apr_mu() (*in module pyttb.cp_apr*), 64
tt_cp_apr_pdnr() (*in module pyttb.cp_apr*), 65
tt_cp_apr_pqnr() (*in module pyttb.cp_apr*), 66
tt_from_sparse_matrix() (*in module pyttb.sptensor*), 32
tt_linesearch_prowsubprob() (*in module pyttb.cp_apr*), 68
tt_loglikelihood() (*in module pyttb.cp_apr*), 70
tt_loglikelihood_row() (*in module pyttb.cp_apr*), 69
tt_to_sparse_matrix() (*in module pyttb.sptensor*), 32
ttensor (*class in pyttb.ttensor*), 56
ttm() (*pyttb.sptensor.sptensor method*), 42
ttm() (*pyttb.tensor.tensor method*), 51
ttm() (*pyttb.ttensor.ttensor method*), 59
ttsv() (*pyttb.tensor.tensor method*), 51
ttt() (*pyttb.tensor.tensor method*), 51

ttv() (*pyttb.ktensor method*), 26
ttv() (*pyttb.sptensor.sptensor method*), 38
ttv() (*pyttb.tensor.tensor method*), 51
ttv() (*pyttb.ttensor.ttensor method*), 58
tucker_als() (*in module pyttb.tucker_als*), 72

U

update() (*pyttb.ktensor method*), 28

V

vectorizeForMu() (*in module pyttb.cp_apr*), 71