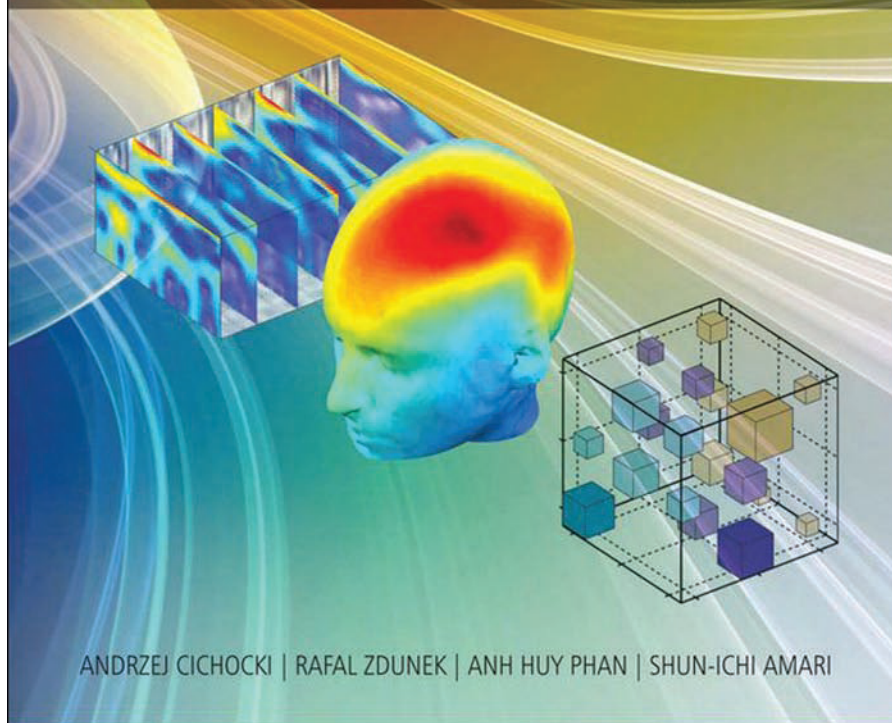


Nonnegative Matrix and Tensor Factorizations

Applications to Exploratory Multi-way
Data Analysis and Blind Source Separation



 WILEY

TensorLABsp Documentation

Release 0.1

Willy DUVILLE

April 19, 2010

CONTENTS

1	Basis of Tensor Algebra	3
1.1	Matrix Multiplication	3
1.1.1	Hadamard product	3
1.1.2	Kroneker product	4
1.1.3	Khatri-Rao product	4
1.1.4	Matricized tensor times Khatri-Rao product	5
1.2	Tensor	5
1.2.1	Outer product	5
1.2.2	Contracted Tensor Product — Tensor times Tensor	6
1.2.3	mode-n Tensor-Matrix product — Tensor times Matrix	6
1.2.4	mode-n Tensor-Vector product — Tensor times Vector	7
1.2.5	Frobenius norm	8
1.2.6	Inner product — scalar product	9
1.2.7	Vectorization	10
1.2.8	Matricization — Unfolding	10
1.2.9	N-vect	12
2	Nonnegative Matrix Factorisation	15
2.1	Hierachical Alternating Least Squares	15
2.1.1	Fast HALS	15
2.1.2	Fast beta HALS	15
3	Tensor Decomposition with PARAFAC Model	17
3.1	Cadecomp-PARAFAC model	17
4	Tensor Factorisation with Tucker Model	19
4.1	Tucker model	19
5	About — Draft & Tests	21
5.1	Testings	23
5.2	Todo list	23
5.3	Glossary	24

5.4 Acknowledgements	24
6 Indices and tables	25
Index	27

Project TensorLABsp

Authors Willy Duville / Ahn Huy Phan / Andrzej Cichocki

Version 0.1.0415

Source bitbucket.org/Wiil/tensorlabsp

Bug tracker bitbucket.org/Wiil/tensorlabsp/issues

<p>Warning: This documentation is work-in-progress and unorganized.</p>

BASIS OF TENSOR ALGEBRA

1.1 Matrix Multiplication

Several special matrix products are important for representation of tensor factorizations and decompositions.

1.1.1 Hadamard product

Definition: The Hadamard product of two equal-sized matrices is the **elementwise product** denoted as \otimes and defined as

$$A \otimes B = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1n}b_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \cdots & a_{mn}b_{mn} \end{bmatrix}$$

Matlab example:

```
>> A = [1,2;3,4]; B = [2,1;1,2];
>> A .* B
```

```
ans =
```

```
     2     2
     3     8
```

Python example:

```
>>> A = array([[1,2],[3,4]]); B = array([[2,1],[1,2]])
>>> A * B
array([[2, 2],
       [3, 8]])
```

1.1.2 Kroneker product

Definition: The Kronecker product of two matrices $A \in \mathbb{R}^{I \times J}$ and $B \in \mathbb{R}^{T \times R}$ is a matrix denoted as $A \otimes B \in \mathbb{R}^{IT \times JR}$ and defined as

$$A \otimes B = \begin{bmatrix} a_{11} B & a_{12} B & \cdots & a_{1J} B \\ a_{21} B & a_{22} B & \cdots & a_{2J} B \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1} B & a_{I2} B & \cdots & a_{IJ} B \end{bmatrix}$$

$$= \begin{bmatrix} a_1 \otimes b_1 & a_1 \otimes b_2 & a_1 \otimes b_3 & \cdots & a_J \otimes b_{R-1} & a_J \otimes b_R \end{bmatrix}.$$

Python function:

<code>numpy.kron</code>	Kronecker product of two arrays.
<code>scipy.linalg.kron</code>	Kronecker product of a and b.

1.1.3 Khatri-Rao product

Definition: For two matrices $A = [a_1, a_2, \dots, a_J] \in \mathbb{R}^{I \times J}$ and $B = [b_1, b_2, \dots, b_J] \in \mathbb{R}^{T \times J}$ with the same number of columns J , their Khatri-Rao product, denoted by \odot , performs the following operation:

$$A \odot B = \begin{bmatrix} a_1 \otimes b_1 & a_2 \otimes b_2 & \cdots & a_J \otimes b_J \end{bmatrix} \\ = \text{vec}(b_1 a_1^T) \text{vec}(b_2 a_2^T) \cdots \text{vec}(b_J a_J^T) \in \mathbb{R}^{IT \times J}$$

Python function:

khattrirao (A, B)

Khatri-Rao product of a and b

Parameters **A** : array, shape (I, J)

B : array, shape (T, J)

Returns **C** : array, shape (I*T, J)

Examples

```
>>> from numpy import array
>>> from tensor import khattrirao
```



```
>>> khatri rao(array([[1,2],[2,1]]), array([[1,1],[1,1]]))
array([[1, 2],
       [1, 2],
       [2, 1],
       [2, 1]])
```

Implementation:

```
from scipy import kron, array, newaxis

C = array([kron(A[:,p][newaxis], B[:,p][newaxis]) for p in range(A.shape[1])])
return C[:,0,:].T
```

1.1.4 Matricized tensor times Khatri-Rao product

Python function:**mttkrp** (*X*, *U*, *n*)

Matricized tensor times Khatri-Rao product for tensor

Calculates the matrix product of the *n*-mode matricization of *X* with the Khatri-Rao product of all entries in *U*, a list of matrices, except the *n*th.

Parameters *X* : Tensor*U* : list of matrices*n* : ...**Returns** *C* : ...

1.2 Tensor

tensor.tensor.ndims	The Tensor's number of dimensions
tensor.tensor.permute	returns a tensor permuted by the order specified.
tensor.tensor.ipermute	returns a tensor permuted by the inverse of the order specified.

1.2.1 Outer product

Definition: The outer product of the tensors $\underline{\mathbf{Y}} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ and $\underline{\mathbf{X}} \in \mathbb{R}^{J_1 \times J_2 \times \dots \times J_M}$ is given by

$$\underline{\mathbf{Z}} = \underline{\mathbf{Y}} \circ \underline{\mathbf{X}} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N \times J_1 \times J_2 \times \dots \times J_M}$$

where

$$z_{i_1, i_2, \dots, i_N, j_1, j_2, \dots, j_M} = y_{i_1, i_2, \dots, i_N} x_{j_1, j_2, \dots, j_M}$$

seealso: `numpy.outer` Compute the outer product of two vectors.

1.2.2 Contracted Tensor Product — Tensor times Tensor

Definition: The contracted product of two tensors $\underline{\mathbf{A}} \in \mathbb{R}^{I_1 \times \dots \times I_M \times J_1 \times \dots \times J_N}$ and $\underline{\mathbf{B}} \in \mathbb{R}^{I_1 \times \dots \times I_M \times K_1 \times \dots \times K_P}$ along the first M modes is a tensor of size $J_1 \times \dots \times J_N \times K_1 \times \dots \times K_P$, given by

$$\langle \underline{\mathbf{A}}, \underline{\mathbf{B}} \rangle_{1, \dots, M; 1, \dots, M}(j_1, \dots, j_N, k_1, \dots, k_P) = \sum_{i_1=1}^{I_1} \dots \sum_{i_M=1}^{I_M} a_{i_1, \dots, i_M, j_1, \dots, j_N} b_{i_1, \dots, i_M, k_1, \dots, k_P}.$$

seealso: `numpy.tensordot` Compute tensor dot product along specified axes for arrays >= 1-D.

1.2.3 mode-n Tensor-Matrix product — Tensor times Matrix

Definition: The mode-n product $\underline{\mathbf{Y}} = \underline{\mathbf{G}} \times_n \mathbf{A}$ of a tensor $\underline{\mathbf{G}} \in \mathbb{R}^{J_1 \times J_2 \times \dots \times J_N}$ and a matrix $\mathbf{A} \in \mathbb{R}^{I_n \times J_n}$ is a tensor $\underline{\mathbf{Y}} \in \mathbb{R}^{J_1 \times \dots \times J_{n-1} \times I_n \times J_{n+1} \times \dots \times J_N}$, with elements

$$y_{j_1, j_2, \dots, j_{n-1}, i_n, j_{n+1}, \dots, j_N} = \sum_{j_n=1}^{J_n} g_{j_1, j_2, \dots, j_N} a_{i_n, j_n}$$

Python function:

ttm (*mat*, *dims=None*, *transpose=False*, *excludedim=False*)

mode-n tensor-matrix product

Parameters **self** : Tensor

mat : ndarray

Single matrix or a list of matrices to be sequentially multiplied along all dimensions.

dims : specifies the dimension (or mode) of X along which mat should be multiplied

option : if 't', performs the same computations as above except the matrices are transposed

excludedim : if True, multiply along all mode but those specified in the dims parameter

Returns \mathbf{a} : ndarray

The filled array.

Implementation:

```

1  ...
2  N = self.ndims();
3  shp = self.shape;
4  order = [dims]+range(0, dims)+range(dims+1, N)
5
6  newdata = self.permute(order)
7  newdata = newdata.data.reshape(shp[dims], self.data.size/shp[dims])
8
9  if transpose:
10     newdata = numpy.dot(mat.transpose(), newdata)
11     p = mat.shape[1]
12 else:
13     newdata = numpy.dot(mat, newdata)
14     p = mat.shape[0]
15
16 newshp = [p] + list(shp[0:dims]) + list(shp[dims+1:N])
17
18 Y = tensor(newdata, newshp)
19 Y = Y.ipermute(order)
20 return Y

```

1.2.4 mode-n Tensor-Vector product — Tensor times Vector

Definition: The mode-n multiplication of a tensor $\underline{\mathbf{Y}} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ by a vector $\mathbf{a} \in \mathbb{R}^{I_n}$ is denoted by ¹

$$\underline{\mathbf{Y}} \bar{\times}_n \mathbf{a}$$

and has dimension $I_1 \times \dots \times I_{n-1} \times I_{n+1} \times \dots \times I_N$, that is,

$$\underline{\mathbf{Z}} = \underline{\mathbf{Y}} \bar{\times}_n \mathbf{a} \in \mathbb{R}^{I_1 \times \dots \times I_{n-1} \times I_{n+1} \times \dots \times I_N},$$

Element-wise, we have

$$z_{i_1, i_2, \dots, i_{n-1}, i_{n+1}, \dots, i_N} = \sum_{i_n=1}^{I_n} y_{i_1, i_2, \dots, i_N} a_{i_n}$$

¹ A bar over the operator \times indicates a contracted product.

Python function:**ttv**(*vect*, *dims*=, [])

mode-n tensor-vector product

Parameters **self** : Tensor, ndims: N**vect** : sequence of vectors**dims** : specifies the dimension (or mode) of X along which vect should be multiplied**Returns** **Y** : Tensor, ndims: N-1**Implementation:**

```
1  ...
2  dims = range(len(vect))
3  vidx = range(len(vect))
4
5  """ Permute it so that the dimensions we're working with come last """
6  remdims = numpy.setdiff1d(range(self.ndims()), dims)
7  if self.ndims() > 1:
8      c = self.permute(numpy.concatenate([remdims, numpy.array(dims)]))
9      c = c.data
10
11  n = self.ndims() - 1
12  for i in range(len(dims) - 1, -1, -1):
13      if n == 0:
14          c = c.reshape( [1, self.shape[n]] )
15      else:
16          c = c.reshape( [reduce(numpy.multiply, self.shape[0:n]), self.shape[n]] )
17      c = numpy.dot(c, vect[vidx[i]])
18      n -= 1
19
20  return c
```

1.2.5 Frobenius norm

Definition:

$$|A|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2} = \sqrt{\text{trace}(A^* A)} = \sqrt{\sum_{i=1}^{\min\{m,n\}} \sigma_i^2}$$

Python function:

norm()

Frobenius norm of a Tensor

seealso: `scipy.linalg.norm` Matrix or vector norm.

norm()

Frobenius norm for a K-Tensor

Implementation

```

1 def norm(self):
2     """ Frobenius norm for a K-Tensor
3     """
4     from numpy import sqrt, outer, dot
5
6     # Compute the matrix of correlation coefficients
7     coefMatrix = outer(self.mylambda, self.mylambda.T)
8     for i in range(self.ndims()):
9         coefMatrix *= dot(self.u[i].T, self.u[i])
10
11     return sqrt(coefMatrix.sum())

```

1.2.6 Inner product — scalar product

Definition: The inner product of two tensors $\underline{\mathbf{A}}, \underline{\mathbf{B}} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ of the same order is denoted by $\langle \underline{\mathbf{A}}, \underline{\mathbf{B}} \rangle$ and is computed as a sum of element-wise products over all the indices, that is

$$c = \langle \underline{\mathbf{A}}, \underline{\mathbf{B}} \rangle = \sum_{i_1}^{I_1} \sum_{i_2}^{I_2} \dots \sum_{i_N}^{I_N} b_{i_1, i_2, \dots, i_N} a_{i_1, i_2, \dots, i_N} \in \mathbb{R}$$

Python function:

innerprod(Y)

Inner product of a tensor and a K-tensor

Parameters **A** : K-Tensor

Y : Tensor

Returns **scalar** : double

Implementation:

```

1 def innerprod(self, Y):
2     """ Inner product of a tensor and a K-tensor
3     """

```

```

4     res = 0
5     for r in range(self.mylambda.size):
6         vect = [n[:,r] for n in self.u]
7         res += self.mylambda[r] * Y.ttv(vect)
8     return res

```

Implementation 2:

```

1 return sum([self.mylambda[r] * Y.ttv([n[:,r] for n in self.u]) for r in range(self.

```

seealso: numpy.inner Inner product of two arrays.

1.2.7 Vectorization

Definition: It is often convenient to represent tensors and matrices as vectors, whereby vectorization of matrix $\mathbf{Y} = [\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_T] \in \mathbb{R}^{I \times T}$ is defined as

$$\mathbf{y} = \text{vec}(\mathbf{Y}) = [\mathbf{y}_1^T, \mathbf{y}_2^T, \dots, \mathbf{y}_T^T]^T \in \mathbb{R}^{IT}.$$

The vec-operator applied on a matrix \mathbf{Y} stacks its columns into a vector. The reshape is a reverse function to vectorization which converts a vector to a matrix.

Example

```

1 >>> from numpy import array
2 >>> v=array([[1,2],[3,4],[5,6]]).flatten()
3 array([1, 2, 3, 4, 5, 6])
4 >>> v.reshape(3, 2)
5 array([[1, 2],
6         [3, 4],
7         [5, 6]])

```

seealso: numpy.ndarray.flatten Return a copy of the array collapsed into one dimension.

See Also:

<http://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html>

1.2.8 Matricization — Unfolding

Definition: The mode- n unfolding of tensor $\underline{\mathbf{Y}} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ is denoted by ² $\mathbf{Y}_{(n)}$ and arranges the mode- n fibers into columns of a matrix. More specifically, a tensor element (i_1, i_2, \dots, i_N)

² We use the Kolda - Bader notations cite{Kolda08}

maps onto a matrix element (i_n, j) , where

$$j = 1 + \sum_{p \neq n} (i_p - 1) J_p, \quad \text{with} \quad J_p = \begin{cases} 1, & \text{if } p = 1 \text{ or if } p = 2 \text{ and } n = 1, \\ \prod_{m \neq n}^{p-1} I_m, & \text{otherwise.} \end{cases}$$

Python function:

matricization (*rdims=None, cdims=None, tsize=None*)

mode-n unfolding of a tensor

reimplementation of the tenmat class

Parameters **K** : Tensor

rdims : int

cdims : list

tsize : int

Returns **M** : Matrix

Implementation:

```

1  def matricization(self, rdims = None, cdims = None, tsize = None):
2      import numpy as np
3
4      if tsize is None: tsize = self.shape
5      nn = np.mgrid[1:self.ndims()+1]
6
7      if bool(rdims) ^ bool(cdims):
8          if rdims is None: rdims = np.setdiff1d(nn, np.array(cdims))
9          if cdims is None: cdims = np.setdiff1d(nn, np.array(rdims))
10
11     else:
12         raise ValueError("You have to specify either rdims or cdims")
13     rdims = np.array(rdims, ndmin = 1)
14     cdims = np.array(cdims, ndmin = 1)
15     rcdims = np.hstack((rdims, cdims))
16
17     if np.setdiff1d(rcdims, nn) or np.setdiff1d(nn, rcdims):
18         raise ValueError("Incorrect specification of dimensions." +
19                             "\n RDIMS = %s\n CDIMS = %s" % (rdims, cdims))
20
21     T = self.permute(rcdims-1);
22
23     row = reduce(np.multiply, [tsize[i-1] for i in rdims])

```

```
24         col = reduce(np.multiply, [tsize[i-1] for i in cdims])
25
26     return T.data.reshape([row, col])
```

1.2.9 N-vect

Python function:

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

Compute the leading mode-n vectors for a tensor

computes the *r* leading eigenvalues of $X_n * X_n'$ (where X_n is the mode-n matricization of X), 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.

Parameters **X** : Tensor

n : int, mode-n matricization of X

r : int, nnumber of leading eigenvalues to return

flipsign : bool, make each column's largest element positive / Make the largest magnitude element be positive

Returns **M** : Matrix

Implementation:

```
1  from numpy import dot
2  from scipy.sparse.linalg.eigen.arpack import eigen_symmetric
3
4  Xn = self.matricization(n)
5  Y = dot(Xn, Xn.T)
6
7  v = eigen_symmetric(Y, r, which = 'LM')
8
9  if flipsign:
10     """ not implemented """
11     pass
12  return v[1]
```

seealso:

<code>scipy.sparse.linalg.eigen.arpack.eigen_symmetric</code>	Finds eigenvalues and eigenvectors of the real symmetric
<code>scipy.sparse.linalg.eigen.arpack</code>	Eigenvalue solver using iterative methods.

Functions summary

<code>tensor.khatrirao</code>	Khatri-Rao product of a and b
<code>tensor.mttkrp</code>	Matricized tensor times Khatri-Rao product for tensor
<code>tensor.tensor.nvecs</code>	Compute the leading mode-n vectors for a tensor
<code>tensor.tensor.matricization</code>	mode-n unfolding of a tensor
<code>tensor.tensor.ttm</code>	mode-n tensor-matrix product
<code>tensor.tensor.ttv</code>	mode-n tensor-vector product
<code>tensor.tensor.norm</code>	Frobenius norm of a Tensor
<code>tensor.tensor.ndims</code>	The Tensor's number of dimensions
<code>tensor.tensor.permute</code>	returns a tensor permuted by the order specified.
<code>ktensor.ktensor.norm</code>	Frobenius norm for a K-Tensor
<code>ktensor.ktensor.innerprod</code>	Inner product of a tensor and a K-tensor

Functions from Python Libraries

<code>numpy.inner</code>	Inner product of two arrays.
<code>numpy.outer</code>	Compute the outer product of two vectors.
<code>numpy.tensordot</code>	Compute tensor dot product along specified axes for arrays ≥ 1 -D.
<code>numpy.ndarray.flatten</code>	Return a copy of the array collapsed into one dimension.
<code>numpy.kron</code>	Kronecker product of two arrays.
<code>scipy.linalg.kron</code>	Kronecker product of a and b.
<code>scipy.linalg.norm</code>	Matrix or vector norm.
<code>scipy.sparse.linalg.eigen.arpack</code>	Find k eigenvalues and eigenvectors of the real symmetric
<code>scipy.sparse.linalg.eigen.arpack</code>	Eigenvalue solver using iterative methods.

NONNEGATIVE MATRIX FACTORISATION

Warning: Documentation Under Construction

2.1 Hierarchical Alternating Least Squares

2.1.1 Fast HALS

2.1.2 Fast beta HALS

TENSOR DECOMPOSITION WITH PARAFAC MODEL

Warning: Documentation Under Construction

A key feature of the analysis of three-way arrays by Candecomp/Parafac is the essential uniqueness of the trilinear decomposition. Kruskal has previously shown that the three component matrices involved are essentially unique when the sum of their k-ranks is at least twice the rank of the decomposition plus 2. It was proved that Kruskal's sufficient condition is also necessary when the rank of the decomposition is 2 or 3. If the rank is 4 or higher, the condition is not necessary for uniqueness. However, when the k-ranks of the component matrices equal their ranks, necessity of Kruskal's condition still holds in the rank-4 case. Ten Berge and Sidiropoulos conjectured that Kruskal's condition is necessary for all cases of rank 4 and higher where ranks and k-ranks coincide. In the present paper we show that this conjecture is false. <http://portal.acm.org/citation.cfm?id=1647955.1648010>

3.1 Candecomp-PARAFAC model

Python function:

```
cp_als (X, R, fitchangetol=1.0000000000000001e-05, maxiters=200, verbose=1, init='eigs')  
CP_ALS
```

Compute a CP decomposition of any type of tensor.

The PARAFAC can be formulated as follows (see Figures 1.26 and 1.27 for graphical representations). Given a data tensor $Y \in \mathbb{R}^{I \times J \times Q}$ and the positive index J , find three-component matrices, also called loading matrices or factors, $A = [a_1, a_2, \dots, a_J] \in \mathbb{R}^{I \times J}$, $B = [b_1, b_2, \dots, b_J] \in \mathbb{R}^{J \times T}$ and $C = [c_1, c_2, \dots, c_J] \in \mathbb{R}^{Q \times J}$ which perform the following approximate factorization:

$$Y = \sum_{j=1}^J a_j \otimes b_j \otimes c_j + E = A, B, C + E, \quad (1.123)$$

or equivalently in the element-wise form (see Table 1.2 for various representations of PARAFAC)

$$y_{itq} = \sum_{j=1}^J a_{ij} b_{tjcqj} + e_{itq}. \quad (1.124)$$

TENSOR FACTORISATION WITH TUCKER MODEL

Warning: Documentation Under Construction

Two of the most commonly used decompositions are the Tucker decomposition and PARAFAC (also known as CANDECOMP or simply CP) which are often considered (thought of) as higher-order generalizations of the matrix singular value decomposition (SVD) or principal component analysis (PCA).

- Cadecomp-*PARAFAC* model
- Tucker model
- nmf ntd ntf

```
1 >>> for i in range(10):
2     ...     print i
```

$$Y = AX + E = ABT + E$$

4.1 Tucker model

Python function:

```
tucker_als(X, R, fitchangetol=1.0000000000000001e-05, maxiters=200, verbose=1,
            init='eigs')
```

The original Tucker model makes the assumption of orthogonality of the factor matrices (in analogy to SVD), [17,79,83,84,107,117]. We will, however, ignore these constraints. By imposing nonnegativity constraints the problem of estimating the component matrices and a core tensor is converted into a generalized NMF problem called the Nonnegative Tucker Decomposition (NTD) (see Chapter 7 for details). The first implementations of Tucker decomposition with nonnegativity constraints together with a number of other constraints were

given by Kiers, Smilde and Bro in [17,79]. The NTD imposes nonnegativity constraints for all component matrices and a core tensor, while a semi-NTD (in analogy to semi-NMF) imposes nonnegativity constraints to only some components matrices and/or some elements of the core tensor.

class `ttensor` (*core, uIn*)

`copy()`

`size()`

`totensor()`

returns a tensor object that is represented by the tucker tensor

`tucker_als` (*X, R, fitchangetol=1.0000000000000001e-05, maxiters=200, verbose=1, init='eigs'*)

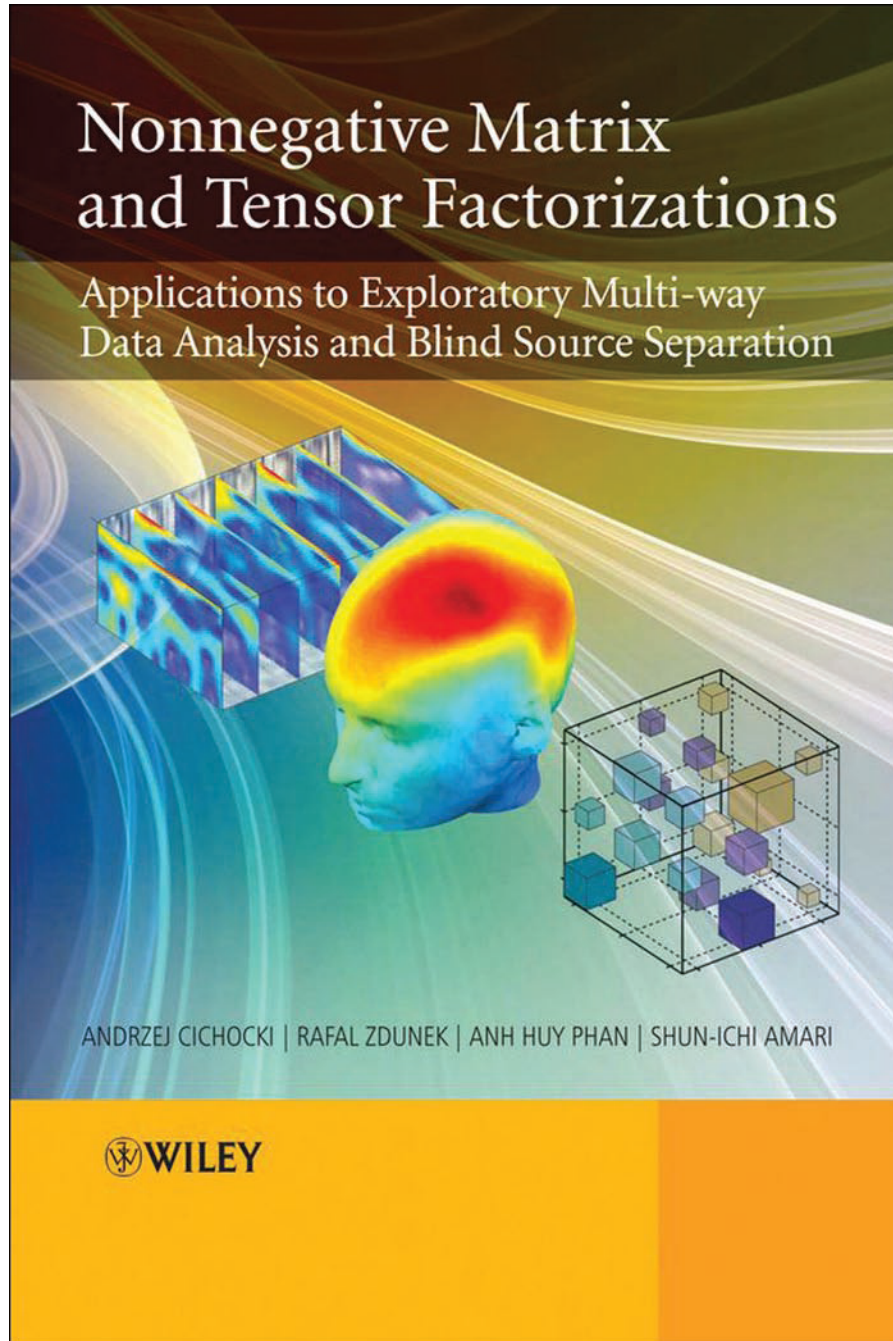
ABOUT — DRAFT & TESTS

Maintainer Willy Duville

Authors Willy Duville <wduville@brain.riken.jp>, Ahn Huy Phan
<phan@brain.riken.jp>, Andrzej Cichocki <cia@brain.riken.jp>

Website <http://www.bsp.brain.riken.jp>

Many modern applications generate large amounts of data with multiple aspects and high dimensionality for which tensors (i.e., multi-way arrays) provide a natural representation.



1. This project is developed in the [Laboratory for Advanced Brain Signal Processing](#), [RIKEN Brain Science Institute](#)
2. Large parts of this documentation originate from Andrzej CICHOCKI's book [Nonnegative Matrix and Tensor factorization](#).

5.1 Testings

Operation	Result	Notes
x or y	if x is <i>environment</i> , then y, else x	(1)
x and y	if x is false, then x, else y	(2)
not x	if x is false, then True, else False	(3)

$$y = ax^2 + bx + c \quad (5.1)$$

$$f(x) = x^2 + 2xy + y^2 \quad (5.2)$$

Imports:

```
.. literalinclude:: ../../src/tensor.py
    :pyobject: khatrirao
    :language: python
    :linenos:
    :start-after: #!End
```

5.2 Todo list

Documentation

- Solve the conflict between *begin{gather}* *begin{split}* and *begin{eqnarray}* on the :math: extention <<http://sphinx.pocoo.org/latest/ext/math.html>> : nowarp:?
- Completion of the Index
- Completion of the Module page
- Feed the glossary
- Write functions parameters definition
- NMF documentation
- NTF documentation
- NTD documentation
- MTTKRP documentation
- Nvec documentation
- Setup the doctest extention <http://docs.python.org/library/doctest.htm>

Programmation

- **Proper testing sets**
 - Check Mathematic properties
 - Check consistency with existing matlab code
- Visualisation for Benchmarks

+++

5.3 Glossary

Numpy <http://www.numpy.org>

Scipy Python's Open Source Library of Scientific Tools. www.scipy.org

environment A structure where information about all documents under the root is saved, and used for cross-referencing. The environment is pickled after the parsing stage, so that successive runs only need to read and parse new and changed documents.

source directory The directory which, including its subdirectories, contains all source files for one Sphinx project.

5.4 Acknowledgements

- Many of those algorithms were borrowed or inspired from the work of KOLDA et. al. ...
- This project meant to implement algorithms of the *Nonnegative Matrix and Tensor Factorizations* book by...

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

INDEX

CADECOMP, 17
Canonical Decomposition, 17
copy() (ttensor.ttensor method), 20
cp_als() (in module ktensor), 17

environment, 24

Harsmann, 17

innerprod() (ktensor.ktensor method), 9

khatrirao() (in module tensor), 4
Kruskal Tensor, 17

matricization() (tensor.tensor method), 11
mttkrp() (in module tensor), 5

norm() (ktensor.ktensor method), 9
norm() (tensor.tensor method), 8
Numpy, 24
nvecs() (tensor.tensor method), 12

Parallel Factor Analysis
 PARAFAC, 17

Scipy, 24
size() (ttensor.ttensor method), 20
source directory, 24

totensor() (ttensor.ttensor method), 20
ttensor (class in ttensor), 20
ttensor (module), 20
ttm() (tensor.tensor method), 6
ttv() (tensor.tensor method), 8
tucker_als() (in module ttensor), 19, 20