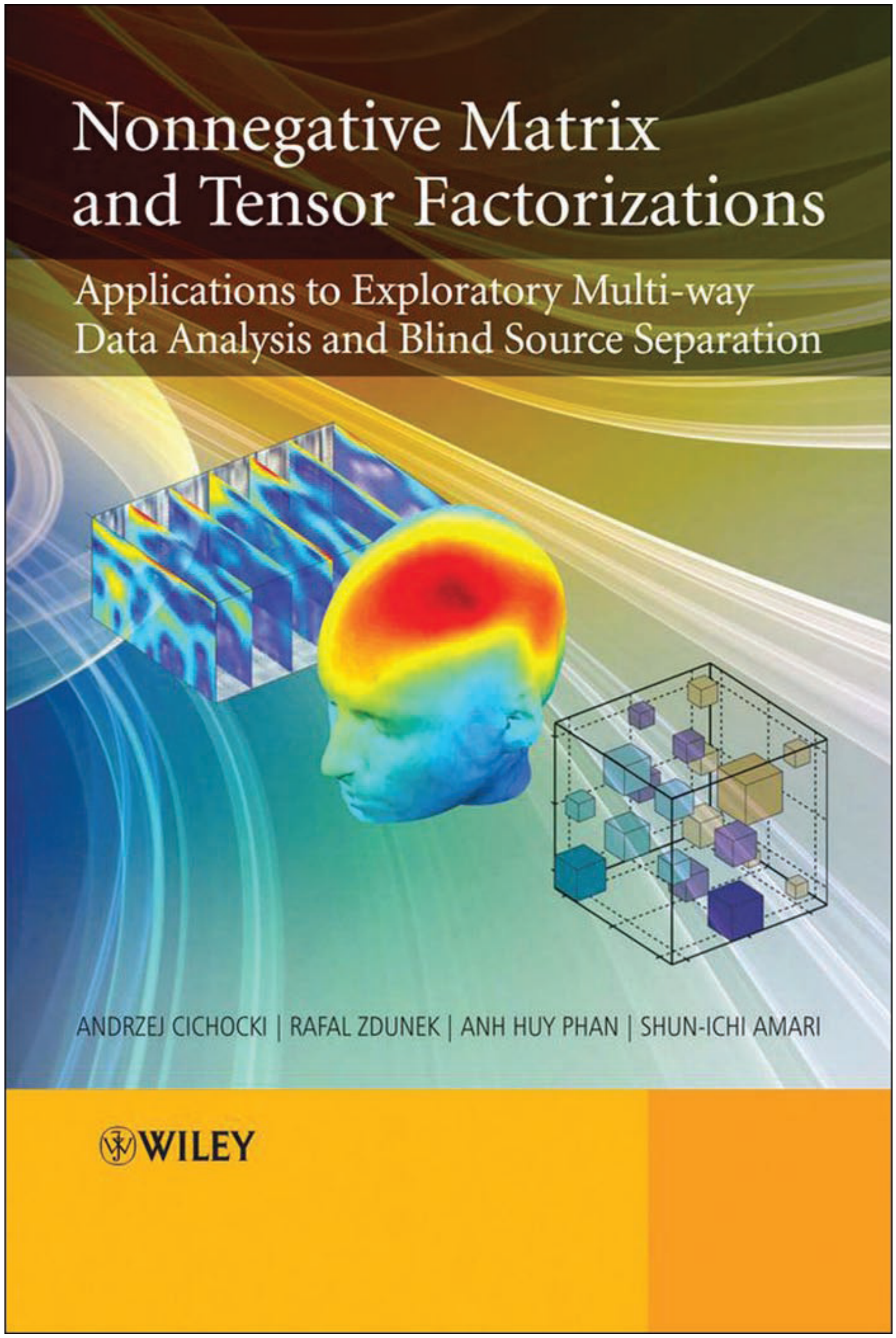


TensorLABsp Documentation

Release 0.1

Willy DUVILLE

April 15, 2010

CONTENTS

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

Warning: This documentation is work-in-progress and unorganized.

CHAPTER

ONE

BASIS OF TENSOR ALGEBRA

Content

1.1 Matrix Multiplication

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

1.1.1 Hadamard product

The Hadamard product of two equal-sizematrices is the **elementwise product** denoted as $\circledast$ and defined as

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

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

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

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

$$= \left[\ \boldsymbol{a}_1 \otimes \boldsymbol{b}_1 \quad \boldsymbol{a}_1 \otimes \boldsymbol{b}_2 \quad \boldsymbol{a}_1 \otimes \boldsymbol{b}_3 \quad \cdots \quad \boldsymbol{a}_J \otimes \boldsymbol{b}_{R-1} \quad \boldsymbol{a}_J \otimes \boldsymbol{b}_R\ \right].$$

Python function:

`numpy.kron`	Kronecker product of two arrays.
`scipy.linalg.kron`	Kronecker product of a and b.

1.1.3 Khatri-Rao product

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

$$A \odot B = \begin{bmatrix} a_1 \otimes b_1 & a_2 \otimes b_2 & \cdots & a_J \otimes b_J \end{bmatrix}$$
$$= \mathrm{vec}(\boldsymbol{b}_1\boldsymbol{a}_1^T) \ \ \mathrm{vec}(\boldsymbol{b}_2\boldsymbol{a}_2^T) \ \ \cdots \ \ \mathrm{vec}(\boldsymbol{b}_J\boldsymbol{a}_J^T) \in \mathbb{R}^{IT\times J}$$

Python function:

`khatrirao`(*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 khatrirao
>>> khatrirao(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 nth.

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

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

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

where

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

seealso:

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

1.2.2 Contracted Tensor Product — Tensor times Tensor

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

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

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

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

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

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 a : ndarray

The filled array.

Implementation:

```
...
N = self.ndims();
shp = self.shape;
order = [dims]+range(0, dims)+range(dims+1, N)

newdata = self.permute(order)
newdata = newdata.data.reshape(shp[dims], self.data.size/shp[dims])

if transpose:
        newdata = numpy.dot(mat.transpose(), newdata)
        p = mat.shape[1]
else:
        newdata = numpy.dot(mat, newdata)
        p = mat.shape[0]

newshp = [p] + list(shp[0:dims]) + list(shp[dims+1:N])

Y = tensor(newdata, newshp)
Y = Y.ipermute(order)
return Y
```

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

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

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

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

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

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

Element-wise, we have

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

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:

```
...
dims = range(len(vect))
vidx = range(len(vect))

""" Permute it so that the dimensions we're working with come last """
remdims = numpy.setdiff1d(range(self.ndims()), dims)
if self.ndims() > 1:
        c = self.permute(numpy.concatenate([remdims, numpy.array(dims)]))
        c = c.data

n = self.ndims() - 1
for i in range(len(dims) - 1, -1, -1):
        if n == 0:
                c = c.reshape( [1, self.shape[n]] )
        else:
                c = c.reshape( [reduce(numpy.multiply, self.shape[0:n]), self.shape
        c = numpy.dot(c, vect[vidx[i]])
        n -= 1

return c
```

1.2.5 Frobenius norm

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

Python function:

norm()

Frobenius norm of a Tensor

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

norm()

Frobenius norm for a K-Tensor

Implementation

```python
def norm(self):
    """ Frobenius norm for a K-Tensor
    """
    from numpy import sqrt, outer, dot

    # Compute the matrix of correlation coefficients
    coefMatrix = outer(self.mylambda, self.mylambda.T)
    for i in range(self.ndims()):
        coefMatrix *= dot(self.u[i].T, self.u[i])

    return sqrt(coefMatrix.sum())
```

1.2.6 Inner product — scalar product

The inner product of two tensors $\underline{\mathbf{A}}, \underline{\mathbf{B}} \in \mathbb{R}^{I_1 \times I_2, \times \cdots \times I_N}$ 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

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

Python function:

innerprod(Y)

Inner product of a tensor and a K-tensor

Parameters **A** : K-Tensor

Y : Tensor

Returns **scalar** : double

Implementation:

```
def innerprod(self, Y):
  """ Inner product of a tensor and a K-tensor
  """
  res = 0
  for r in range(self.mylambda.size):
    vect = [n[:,r] for n in self.u]
    res += self.mylambda[r] * Y.ttv(vect)
  return res
```

Implementation 2:

```
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

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

$$\boldsymbol{y} = \mathrm{vec}(\mathbf{Y}) = \left[\boldsymbol{y}_1^T, \boldsymbol{y}_2^T, \ldots, \boldsymbol{y}_T^T\right]^T \in \mathbb{R}^{IT}.$$

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

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

The mode-n unfolding of tensor $\underline{\mathbf{Y}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is denoted by [2] $\mathbf{Y}_{(n)}$ and arranges the mode-n fibers into columns of a matrix. More specifically, a tensor element $(i_1, i_2, \ldots, i_N)$ maps onto a matrix element (i_n, j), where

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

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:

```python
def matricization(self, rdims = None, cdims = None, tsize = None):
        import numpy as np

        if tsize is None: tsize = self.shape
        nn = np.mgrid[1:self.ndims()+1]

        if bool(rdims) ^ bool(cdims):
                if rdims is None: rdims = np.setdiff1d(nn, np.array(cdims))
                if cdims is None: cdims = np.setdiff1d(nn, np.array(rdims))

        else:
                raise ValueError("You have to specify either rdims or cdims")
        rdims = np.array(rdims, ndmin = 1)
        cdims = np.array(cdims, ndmin = 1)
        rcdims = np.hstack((rdims, cdims))

        if np.setdiff1d(rcdims, nn) or np.setdiff1d(nn, rcdims):
```

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

```
                raise ValueError("Incorrect specification of dimensions." +
                                 "\n RDIMS = %s\n CDIMS = %s" % (rc

        T = self.permute(rcdims-1);

        row = reduce(np.multiply, [tsize[i-1] for i in rdims])
        col = reduce(np.multiply, [tsize[i-1] for i in cdims])

        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 Xn*Xn' (where Xn 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:

```
from numpy import dot
from scipy.sparse.linalg.eigen.arpack import eigen_symmetric

Xn = self.matricization(n)
Y = dot(Xn, Xn.T)

v = eigen_symmetric(Y, r, which = 'LM')

if flipsign:
        """ not implemented """
        pass
return v[1]
```

seealso:

`scipy.sparse.linalg.eigen.arpack.eigen_symmetric`	Find k eigenvalues and eigenvectors of the real symmetric
`scipy.sparse.linalg.eigen.arpack`	Eigenvalue solver using iterative methods.

Functions from Python

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

Functions from LABSP

`tensor.khatrirao`	Khatri-Rao product of a and b
`tensor.mttkrp`	Matricized tensor times Khatri-Rao product for tensor
`tensor.tensor.nvecs`	Compute the leading mode-n vectors for a tensor
`tensor.tensor.matricizatio`	mode-n unfolding of a tensor
`tensor.tensor.ttm`	mode-n tensor-matrix product
`tensor.tensor.ttv`	mode-n tensor-vector product
`tensor.tensor.norm`	Frobenius norm of a Tensor
`ktensor.ktensor.norm`	Frobenius norm for a K-Tensor
`ktensor.ktensor.innerprod`	Inner product of a tensor and a K-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.

CHAPTER

TWO

NONNEGATIVE MATRIX FACTORISATION

Warning: Documentation Under Construction

2.1 Hierachical Alternating Least Squares

2.1.1 Fast HALS

2.1.2 Fast beta HALS

CHAPTER

THREE

TENSOR DECOMPOSITION WITH PARAFAC MODEL

Warning: Documentation Under Construction

Keywords: Canonical Decomposition, Parallel Factor Analysis, CADECOMP, PARAFAC, Harsmann, Kruskal Tensor

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 Cadecomp-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 YeRIxTxQ and the positive index J, find three-component matrices, also called loading matrices or factors, A = [a1, a2, . . . , aJ] RIJ , B = [b1, b2, . . . , bJ] RTJ and C = [c1, c2, . . . , cJ] RQJ which perform the following approximate factorization:

Y = J j=1 aj ? bj ? cj + E = A, B,C + E, (1.123)

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

yitq =J j=1 aijbtjcqj + eitq. (1.124)

CHAPTER

FOUR

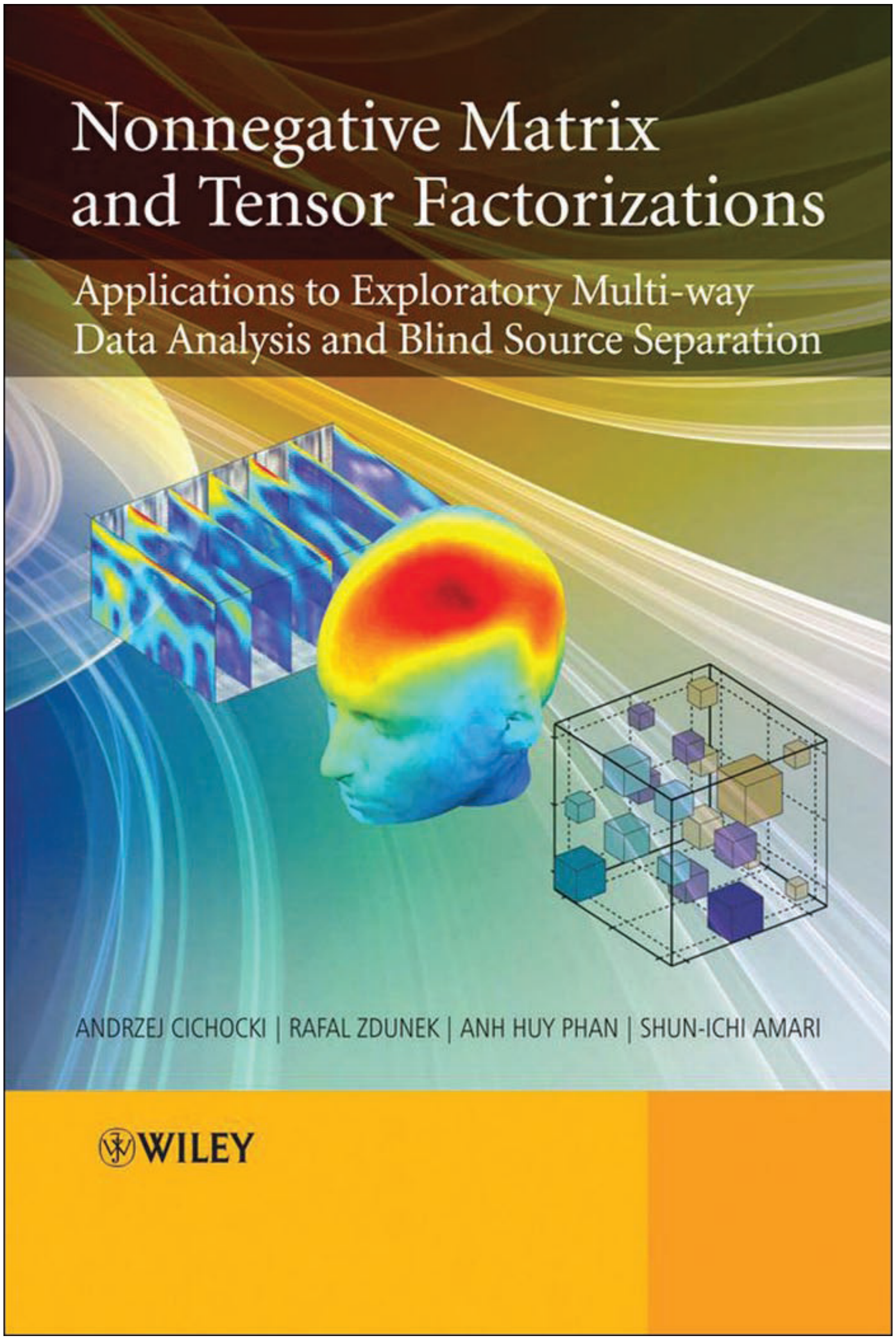
ABOUT — DRAFT & TESTS

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 developped 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.

4.1 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.

4.2 Testings

Operation	Result	Notes
`x or y`	if *x* is *environment*, 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)

Todo list

!end Todo list

eqnarray

$$y = ax^2 + bx + c \tag{4.1}$$

$$f(x) = x^2 + 2xy + y^2 \tag{4.2}$$

Imports:

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

4.3 Acknowledgements

CHAPTER

FIVE

INDICES AND TABLES

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

INDEX