# SPARSE TENSORS DECOMPOSITION SOFTWARE

Papa S Diaw, Master's Candidate

Dr. Michael W. Berry, Major Professor

Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville

July 16th, 2010

## Abstract

The purpose of this project was to build a software package that could be used as an alternative to the MATLAB Tensor Toolbox for high-order sparse tensor parallel factor analysis. The software package will also be used as a tool to incorporate nonnegative tensor factorization into a text visualization environment called *FutureLens*. The project was written in Python, an interpreted, object-oriented language. To test the software and evaluate its performance, the IEEE VAST 2007 contest dataset was used along with the Python profiler.

# 1 Background and Introduction

Modern Internet traffic, telecommunication records, social networks (such as Facebook, MySpace) and Internet Relays Chats produce vast amounts of data with multiple aspects and high dimensionality [3]. Methods to analyze such data are required in areas such as image analysis or text mining. Until recently, Nonnegative Matrix Factorization (NMF) has been a predominant classification technique in the field of text mining. NMF has performed well when dealing with high-dimensional data. NMF models and techniques are capable of providing new insights and relevant information on the hidden relationships in large datasets. However, NMF techniques preprocess multi-way data and arrange them into a matrix. This preprocessing phase requires large computation memory and higher CPU efficiency to accomplish the computations. Moreover, NMF relies on linear relationships in the matrix representation of the data. This representation sometimes fails to capture important structure information. Finally, the two-dimensional matrix calculations can be slower and less accurate. To overcome these issues, scientists are turning to Nonnegative Tensor Factorizations (NTF). The interest in tensors arises from their capability to provide a natural way of dealing with high dimensionality and conserve the original multi-way structure of the data. Tensors (and nonnegative tensor factorizations) are used in a variety of disciplines in the sciences and engineering. Tensor-based techniques "have a wide range of important applications in fields such as in bioinformatics, neuroscience, image processing, text mining, chemo-metrics, computer vision and graphics, as well as any other field where tensor factorizations and decompositions can be used to perform factor retrieval, dimensionality reduction, to mention but a few" [3].

Sandia National Laboratories has developed a Tensor Toolbox for MATLAB, which handles NTF. For more information about the MATLAB tensor toolbox see [1]. Unfortunately, MATLAB licenses are expensive and the Tensor Toolbox for MATLAB may not be getting a lot of exposure outside of academia. Another drawback to using MATLAB is the fact that it is proprietary software. In other words, the users are restricted from modifying or distributing the code.

The goal of this PILOT is to develop a Python software package for nonnegative sparse tensor factorizations based on the MATLAB implementation of the PARAFAC algorithm. The software will provide an alternative to the Tensor Toolbox for MATLAB and will be freely available to anyone. In addition, it will facilitate the incorporation of NTF into FutureLens, which is an interface to explore and visualize features in collections of text documents [17]. Moreover, making the software freely available will give more exposure to NTF and spark more interest in the open source community.

The remainder of this report is arranged as follows: Section 2 gives a general overview of tensors, Section 3 gives a brief overview of nonnegative tensor factorization and PARAFAC, and Section 4 describes the software implementation. Section 5 discusses the performance analysis of the software. Section 6 discusses issues encountered with floating-point arithmetic and convergence. Finally, concluding remarks and a description of future work are given in Section 7.

# 2 Tensors

A tensor is a multi-way array or multi-dimensional matrix. The number of dimensions of a tensor defines its order also known as mode or ways. For example, a third-order tensor has three modes (or indices or dimensions). Scalars, vectors and matrices are special cases of tensors: a scalar is a zero-order tensor, a vector is a first-order tensor, and a matrix is a second-order tensor. When a tensor has three or more dimensions, it is called a high-order tensor. For a complete review of tensors, see Kolda and Bader [2].

In this PILOT, we focus our attention on non-negative sparse tensors. Our interest in the non-negativity of the data is due to the fact that many real-world data are nonnegative and the hidden components have a physical meaning only when nonnegative [3]. For instance, in image processing pixels are represented by nonnegative numbers or in information retrieval, documents are usually represented as relative frequencies of words in a dictionary. In addition, the sparseness allows for features selection and generalizations. For instance in economics, the "sparseness constraints may increase the efficiency of a portfolio" [3].

We introduce several definitions important to NTF.

**DEFINITION 2.1.** An N-way tensor A is rank-one if it can be written as the outer product of N vectors i.e.,

$$A = a_1 \circ a_2 \circ a_3 \circ ..a_N$$

The symbol "$\circ$" represents the vector outer product. Fig 2.1 gives an example of a rank-one third order tensor [3].

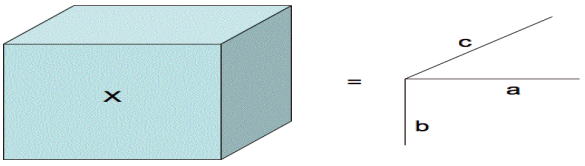


Fig. 2.1 rank-one Third order Tensor

**DEFINITION 2.2**. The outer product of the tensors $Y \in IR^{I_1 \times I_2 \times \ldots I_N}$ and $X \in IR^{J_1 \times J_2 \times \ldots \times J_M}$ is given by

$$Z = Y \circ X \in IR^{I_1 \times I_2 \times \ldots \times I_N \times J_1 \times J_2 \times \ldots \times J_M},$$

where

$$z_{i_1, i_2, \ldots, i_N, j_1, j_2, \ldots, j_M} = y_{i_1, i_2, \ldots, i_N} \times x_{j_1, j_2, \ldots, j_M}$$ [3].

**DEFINITION 2.3.** The rank of a tensor A is defined as the smallest number of rank-one tensors that generates A as their sum [2].

The definition of the rank of a tensor is similar in some respects to the definition of the rank of a matrix. But one difference when it comes to tensors is that, one can have a rank over the rows that is different from the rank over the columns. [2] gives a very detailed explanation of the differences between tensor rank and matrix rank.

**DEFINITION 2.4.** The Kronecker product of two matrices $A \in IR^{I \times J}$ and $B \in IR^{K \times L}$ is given by,

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \dots & a_{1J}B \\ a_{21}B & a_{22}B & \dots & a_{2J}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}B & a_{I2}B & \dots & a_{IJ}B \end{pmatrix} \text{[3].}$$

**DEFINITION 2.5.** For two matrices $A = \left[a_1, a_2, ..., a_J\right] \in IR^{I \times J}$ and $B = \left[b_1, b_2, ..b_J\right] \in IR^{T \times J}$ with the same number of columns J, their Khatri-Rao product, denoted by ⊙, performs the following operation:

$$A \odot B = [a_1 \otimes b_1 \; a_2 \otimes b_2 \cdots a_J \otimes b_J]$$

⊗ represents the Kronecker product [3].

**DEFINITION 2.6.** A tensor fiber is a one-dimensional fragment of a tensor, obtained by fixing all indices except for one [2].

**DEFINITION 2.7.** Unfolding, also known as matricization or flattening, is a process of reordering the elements of an N-th order tensor into a matrix. There are various ways to order the fibers of tensors, which makes the unfolding process not unique [3].

# 3 Tensor Factorizations and PARAFAC

In this section we begin with a brief introduction to tensor factorizations, which is key to our project. Then, we discuss PARAFAC, which is the algorithm used (in this project) to implement the tensor factorizations.

## 3.1 Tensor Factorizations

Tensor factorizations were first introduced by Hitchcock in 1927 and later developed by Cattell in 1944 and Tucker in 1966. The idea behind tensor factorization is to rewrite a given tensor as a finite sum of lower-rank tensors.

The two most popular tensor factorizations models are the Tucker model and the PARAFAC model. In this project, we implemented the PARAFAC to emulate the Tensor Toolbox for MATLAB. The next subsection discusses the PARAFAC. For a review of tensor factorizations, see Kolda and Bader [2].

## 3.2 PARAFAC

A decomposition of a tensor as a sum of rank-one tensors is called PARAFAC (Parallel factor analysis.) PARAFAC is also known as Canonical Decomposition (CANDE-COMPE) (Harsman (1970), Carroll and Chang (1970)) [2].

**DEFINITION 3.1.** Given a three-way tensor **X** and an approximation rank R, we define the factor matrices as the combination of the vectors from the rank-one components.

$$X \approx A \circ B \circ C \approx \sum_{r=1}^{R} a_r \circ b_r \circ c_r \text{ [2].}$$

Fig 3.1 gives an example of a three-way tensor factorizations based on PARAFAC.

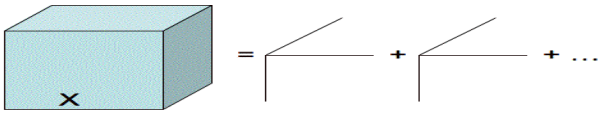


Fig. 3.1 Decomposition of three-way tensor.

In this project we used an Alternating Least Square (ALS) algorithm to implement PARAFAC. The key idea is to minimize the sum of squares between the original tensor and the factorized model of the tensor. As suggested in [7], we cycle "over all the factor matrices and performs a least-square update for one factor matrix while holding all the others constant" [7].

Nonnegative tensor factorization (NTF) is a generalization of nonnegative matrix factorization. NTF can be considered an extension of the PARAFAC model with the constraint of nonnegativity.

# 4 Software Implementation

The following section describes the implementation of the software package. First we present the programming language used in this project. Second we briefly describe the main data structures. Finally we discuss the main components of the software.

## 4.1 Python

Python is the programming language used for this project. Python is an object-oriented, extensible, interpreted language. It runs on essentially all Unix and Linux systems, as well as on DOS/Windows platforms and on the Mac. It is free and comes with complete source code. The choice on using Python is justified by several reasons. First its learning curve is very flat. Anyone with some programming experience can learn how to use python in a few hours. In addition, it supports object methods (everything is an object in Python) and scales nicely. Additionally, there has been a lot of recent interest in Python in the scientific community. Several scientific computing packages such as Numpy [8] and Scipy [9] have been created in order to extend the languages capabilities for scientific programming. Finally, Python is extensible. One can write an object library in C, C++, or native Python, which can then both be dynamically or statically linked with the main Python system and used in Python programs.

## 4.2 Data Structures

Mainly two types of data structure were used to implement the software: dictionaries and Numpy arrays.

### 4.2.1 Python Dictionaries

Python dictionaries are mainly used to store the tensor data. A dictionary in Python is a mutable type of container that can store any number of Python objects, including other container types. Dictionaries consist of pairs of keys and their corresponding values. The structure of the Python dictionaries allows us to exploit the sparseness of our tensors by storing them in a more efficient way. They help the performance (both time and memory). For instance, in the tests we conducted, the tensor obtained from the VAST data set had 1,385,205,184 elements, with 1,184,139 nonzero elements. Our software only stores the nonzero elements and keeps track of the zeros by using the default value of the dictionary (another feature of Python dictionary).

### 4.2.2 Numpy arrays

"Numpy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays" [10]
Numpy arrays were used to perform operations such as the Khatri-Rao products or tensors multiplications (tensors times vectors and matrices times Khatri-Rao product). Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences. In addition, we opted (after profiling and optimization) to store the indices of the tensor in Numpy arrays, which were stored in the dictionary. The choice of Numpy arrays was mainly for speed, since they are already optimized for speed.

## 4.3 Main Modules

In this subsection, we present a brief description of the main modules used (by module, we mean a file containing Python definitions and statements) for our software package. These modules are the core components of our ALS algorithm.

### 4.3.1 SPTENSOR

SPTENSOR is the most important module of our software package. It is defined as a class and allows us to set up the sparse tensors. It takes as arguments the subscripts of the nonzero elements and their values. The subscripts and values could be passed as Numpy arrays, Numpy matrices or Python lists. SPTENSOR transforms the arguments into a dictionary and keeps a few instances variables such as the size, the number of dimensions, and the Frobenius norm.

### 4.3.2 PARAFAC

PARAFAC is the module that coordinates the NTF. It basically implements an Alternating Least Square algorithm. It takes as input the tensor to approximate and the approximation Rank. After the NTF reaches the desired convergence or the maximum numbers of iterations, the factor matrices are turned into a Kruskal Tensor, which is the object returned by the PARAFAC module. Fig. 4.1 gives an illustration of a three-way decomposed tensor. For a detailed review of the ALS used in the PARAFAC module see, Kolda [10].

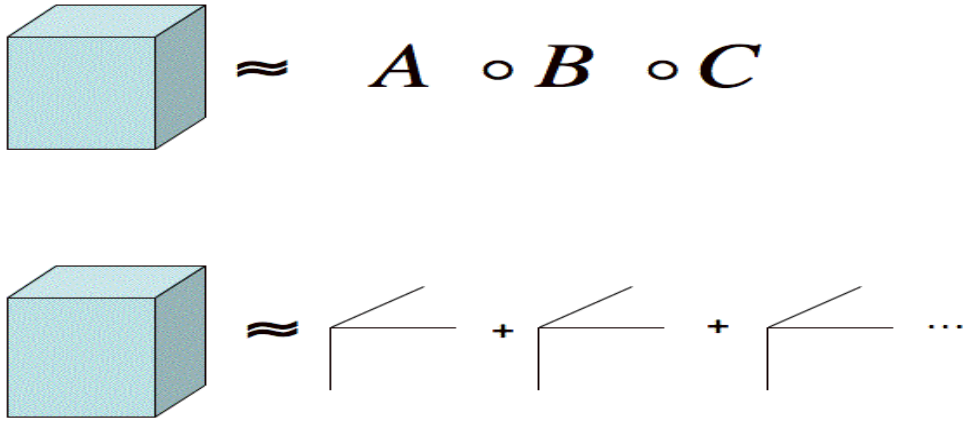


Fig. 4.1 Parafac of a three-way Tensor

### 4.3.3 INNERPROD

INNERPROD efficiently computes the inner product between a SPTENSOR tensor X and KTENSOR Y. It is used by PARAFAC to compute the norm residual, which tells us how close the approximation is to the real value of the tensor.

### 4.3.4 TTV

TTV computes the product of a sparse tensor X with a (column) vector V. This module is the workhorse of our software package. Most of the computation time is spent in this module. It is used by the MTTKRP and INNERPROD modules.

### 4.3.5 MTTKRP

MTTKRP performs two tasks. First it performs the Khatri-Rao product of all factor matrices except the one being updated. Then it performs the matrix multiplication of the matricized version of the tensor being approximated with the Khatri-Rao product obtained above.

### 4.3.6 KTENSOR

KTENSOR creates a Kruskal tensor, which is the object returned after the factorization is done and the factor matrices are normalized. It is defined as class, which keeps some instance variables such as the Norm. The norm of KTENSOR plays a big part in determining the residual norm in the PARAFAC module. Fig. 4.2 gives an illustration of a KTENSOR.

$$lambda = [\lambda a, \lambda_b, \lambda_c]$$

$$U = \left[ \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1r} \\ a_{21} & a_{22} & \cdots & a_{2r} \\ \cdots & \cdots & \ddots & \cdots \\ a_{d_1 1} & a_{d_1 2} & \cdots & a_{d_1 r} \end{bmatrix}, \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1r} \\ b_{21} & b_{22} & \cdots & b_{2r} \\ \cdots & \cdots & \ddots & \cdots \\ b_{d_2 1} & b_{d_2 2} & \cdots & b_{d_2 r} \end{bmatrix}, \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1r} \\ c_{21} & c_{22} & \cdots & c_{2r} \\ \cdots & \cdots & \ddots & \cdots \\ c_{d_3 1} & a_{d_3 2} & \cdots & c_{d_3 r} \end{bmatrix} \right]$$

Fig. 4.2 KTENSOR Representation of a three-Order decomposed Tensor.
The lambda vector represents the absorbed weights from the normalization and U represents the normalized factor matrices.

Fig. 4.3 shows how the modules interact. First an instance of SPTENSOR and a rank approximation are used as inputs. PARAFAC takes the two inputs and starts the factorization. Then during the factorization, PARAFAC calls MTTKRP, INNERPROD, and TTV. Finally when convergence is reached (or maximum number of iterations), a KTENSOR is generated as output.

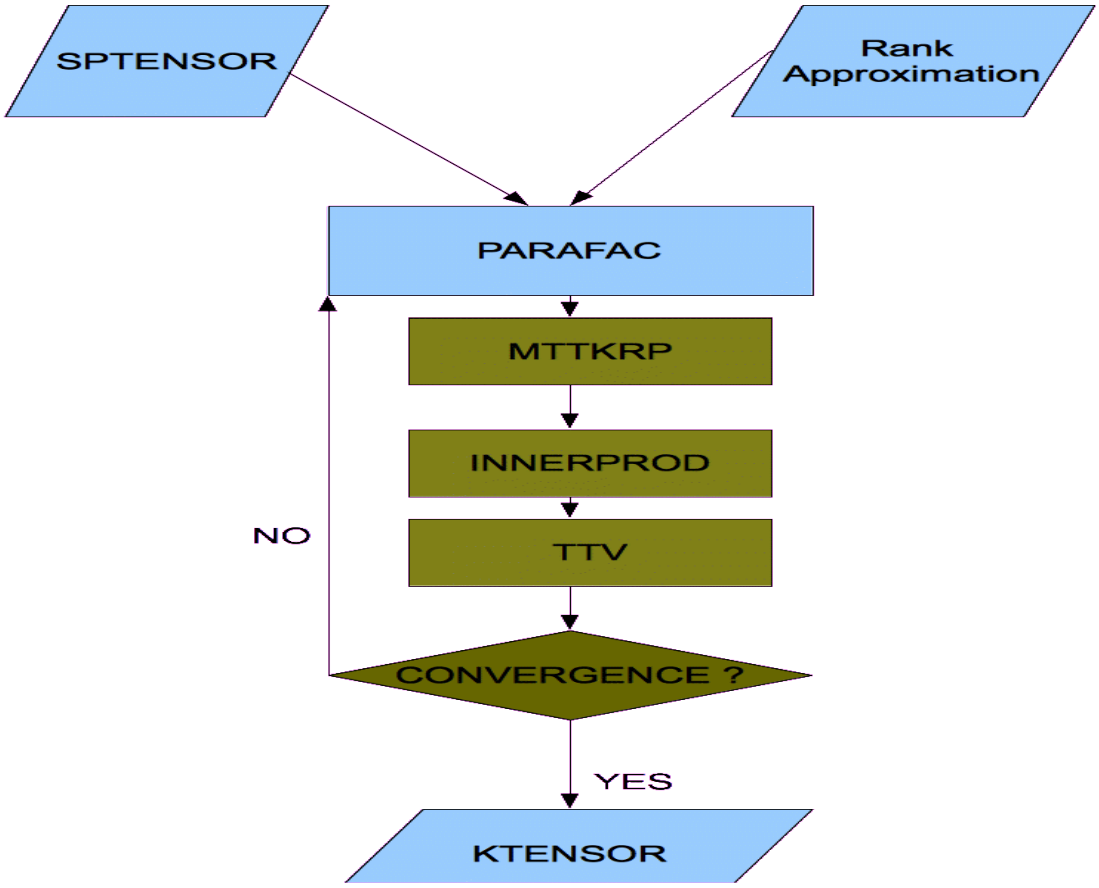


**Fig. 4.3 Interaction of Modules.**

# 5 Performance Analysis

In writing software, "The golden rule is to first write an easy-to-understand program, then verify it, then profile it, and then think about optimization" [13]. While writing our software package, we tried to follow this rule very closely. In this section, we first present the tool used to profile the software. Second, we discuss the bottlenecks identified by the profiler and improvements made.

## 5.1 Profiler

The profiler used for this software package is called the Python cProfile. It is a very important tool for finding bottlenecks in the code. It provides a way of collecting and analyzing statistics about how our different modules consume the processor's resources. Tables 1, 2, and 3 show sample outputs of the Python cProfile. For more detailed coverage of the Python cProfile, see [14].

## 5.2 Bottlenecks and Optimization

In this subsection, we present the bottlenecks and the improvements we added to the codebase to improve the performance of the software. All the tests run during the profiling part used the IEEE VAST 2007 contest dataset.

Initially, Numpy arrays were not used as the internal data structures for the computations, Python lists were. However, the combination of some Numpy functions such as the DOT function or element-wise multiplication function caused us to have to convert Numpy arrays back to Python lists. The conversion from Numpy arrays to Python lists was done by the Numpy built-in function TOLIST. Table 1 (first row), which was obtained using the cProfile, shows that the function call was too expensive. In Table 1, we also noticed that the function RETURN_UNIQUE (second row), which traversed the indices of the tensor and add the values of the duplicated indices, was quite expensive. And Finally, we have the function TTV (Table 1, row 6), which performs the multiplication of a tensor by a vector, was taking more than 11 seconds per call and was making many recursive calls.

| ncalls | Tottime | percall | cumtime | Percall | function |
|---|---|---|---|---|---|
| 2803 | 3605.732 | 1.286 | 3605.732 | 1.286 | tolist |
| 1400 | 1780.689 | 1.272 | 2439.986 | 1.743 | return_unique |
| 9635 | 1538.597 | 0.160 | 1538.597 | 0.160 | array |
| 814018498 | 651.952 | 0.000 | 651.952 | 0.000 | get of 'dict' |
| 400 | 101.308 | 0.072 | 140.606 | 0.100 | setup_size |
| 1575/700 | 81.705 | 0.052 | 7827.373 | 11.182 | ttv |
| 2129 | 39.287 | 0.018 | 39.287 | 0.018 | max |

**Table 1: Modules' Performance when Python lists are used (values are in seconds).**

To fix these problems, we made incremental changes, rerunning the profiler after each one. Table 2 summarizes the performance of the software after we replace the Python lists with the Numpy arrays. In Table 2, we noticed that the Percall times for RETURN_UNIQUE function and TTV both increased. This is characteristic of software optimization. Improving one part of the code may result in performance loss in other areas as evidenced in our case by the functions RETURN_UNIQUE and TTV.

After many improvements such as removing the recursion calls in TTV, removing the RETURN_UNIQUE and replacing it with SETUP_DIC function and creating MYACCUMARRAY function (a poor man's version of MATLAB's ACCUMARRAY [16]), the performance of our software package is summarized in Table 3. We can see that the TTV has improved from 22 to 1.5 seconds. The only function we were not able to improve after many attempts was MYACCUMARRAY.

| ncalls | tottime | percall | cumtime | Percall | function |
|---|---|---|---|---|---|
| 1800 | 15571.118 | 8.651 | 16798.194 | **9.332** | return_unique |
| 12387 | 2306.950 | 0.186 | 2306.950 | 0.186 | array |
| 1046595156 | 1191.479 | 0.000 | 1191.479 | 0.000 | 'get' of 'dict' |
| 1800 | 1015.757 | 0.564 | 1086.062 | 0.603 | setup_size |
| 2025/900 | 358.778 | 0.177 | 20589.563 | 22.877 | ttv |
| 2734 | 69.638 | 0.025 | 69.638 | 0.025 | max |

**Table 2: Modules' Performance when Numpy arrays are used (values in seconds).**

| ncalls | tottime | percall | cumtime | Percall | function |
|---|---|---|---|---|---|
| 75 | 134.939 | 1.799 | 135.569 | 1.808 | myaccumarray |
| 75 | 7.802 | 0.104 | 8.148 | 0.109 | setup_dic |
| 100 | 5.463 | 0.055 | 151.402 | 1.514 | ttv |
| 409 | 2.043 | 0.005 | 2.043 | 0.005 | array |
| 1 | 1.034 | 1.034 | 1.034 | 1.034 | get_norm |
| 962709 | 0.608 | 0.000 | 0.608 | 0.000 | append |
| 479975 | 0.347 | 0.000 | 0.347 | 0.000 | item |
| 3 | 0.170 | 0.057 | 150.071 | 50.024 | mttkrp |
| 25 | 0.122 | 0.005 | 0.122 | 0.005 | sum (Numpy) |
| 87 | 0.083 | 0.001 | 0.083 | 0.001 | dot |

**Table 3: Modules' Performance after Optimization (values in seconds).**

Numpy, which "is the fundamental package for scientific computing in Python"[10], helped us achieved tremendous code acceleration. During the optimization phase, using the Numpy tools allowed us to rewrite code that was more maintainable, which will facilitate any future upgrade or addition.

# 6 Floating-Point arithmetic and Convergence

In this section, we identify two issues we would like to bring to the user's attention. First we discuss a floating-point arithmetic issue in Python, and follow with a related convergence issue.

## 6.1 Floating-point arithmetic

In Python and in MATLAB, "Floating-point numbers are represented in computer hardware as base 2 (binary) fractions", which makes it hard to represent any decimal fraction as an exact binary fraction [19]. In addition to this issue, MATLAB uses double precision whereas Python uses single precision. While comparing, the tensor toolbox For MATLAB to our software, we noticed that the results are not exactly the same when doing the PARAFAC. This discrepancy is attributed to the difference in precision between MATLAB and Python and does not take away anything in the capability of our software. This is just something to keep in mind, when one is comparing our software to the Tensor Toolbox For MATLAB.

Figures 6.1 and 6.2 represent outputs of the PARAFAC of the same sparse tensor respectively in MATLAB and Python.

```
P is a ktensor of size 5 x 7 x 5
P.lambda = [ 1.0255    0.96526    0.004789  0.00038179  7.7472e-05 ]
P.U{1} =
                   0                   0                   0                   0                   0
1.000000000000000  1.000000000000000  0.999999999999997  1.000000000000000  1.000000000000000
0.000000000000000  0.000000000000296  0.000000069693810  0.000000000000000  0.000000000220093
                   0                   0                   0                   0                   0
0.000000000000000  0.000000000000000  0.000000000000000  0.000000000321195  0.000000000007134

P.U{2} =
                   0                   0                   0                   0                   0
0.000000000000000  0.000000000000024  0.000000000000000  0.000000001010566  0.000000339872722
0.999998222609313  0.999999721321112  0.999966834072187  0.998420372451732  0.999887028990220
                   0                   0                   0                   0                   0
0.000875807192780  0.000333326399235  0.000873215054646  0.051822285624710  0.014265197312272
                   0                   0                   0                   0                   0
0.001669652651127  0.000668020366663  0.008097422498258  0.021707385516835  0.004736391309695

P.U{3} =
                   0                   0                   0                   0                   0
1.000000000000000  1.000000000000000  1.000000000000000  1.000000000000000  1.000000000000000
0.000000000000000  0.000000000000000  0.000000000151815  0.000000000000000  0.000000000000000
0.000000000000000  0.000000000000000  0.000000000000000  0.000000000000007  0.000000000418442
0.000000000000000  0.000000000000000  0.000000000000000  0.000000000000000  0.000000000000000
```

Fig. 6.1 MATLAB output of a decomposed tensor

```
ktensor of size [5, 7, 5]
lambda :[  1.02549755944  0.965257000509 0.00478901118102 0.000381792512797 7.74740905878e-05 ]
U:

0.0000000000000000000  0.0000000000000000000  0.0000000000000000000  0.0000000000000000000  0.0000000000000000000
1.0000000000000000000  0.9999999999999966693  0.9999999999671562723  0.9999999999918443017  0.9999999999963251618
0.0000000142774012848  0.0000002328747314042  0.0000256290757187519  0.0000000057550762276  0.0000050554774128818
0.0000000000000000000  0.0000000000000000000  0.0000000000000000000  0.0000000000000000000  0.0000000000000000000
0.0000000000000418390  0.0000000000000048395  0.0000000000245713612  0.0000127706401586963  0.0000069232404755729

0.0000000000000000000  0.0000000000000000000  0.0000000000000000000  0.0000000000000000000  0.0000000000000000000
0.0000000012177930128  0.0000000863055021255  0.0000000125287027935  0.0000604053950829446  0.0012379507371677401
0.9999982265167093160  0.9999997213262696860 2  0.9999668137243419296 9  0.9984181849829512067 6  0.9998844552391490392 3
0.0000000000000000000  0.0000000000000000000  0.0000000000000000000  0.0000000000000000000  0.0000000000000000000
0.0008757845637970792 3  0.0003331907394755001  0.0008737184078767902 8  0.0518608724281367505 8  0.0144343712570039056 6
0.0000000000000000000  0.0000000000000000000  0.0000000000000000000  0.0000000000000000000  0.0000000000000000000
0.0016696391517340202 0  0.0006680163004853114 5  0.0080998806244950288 8  0.0217158414493559236 1  0.0047654771329645115 3

0.0000000000000000000  0.0000000000000000000  0.0000000000000000000  0.0000000000000000000  0.0000000000000000000
1.0000000000000000000  1.0000000000000000000  0.9999999999998034905  0.9999999999999888978  0.9999999993223454009
0.0000000000107582069  0.0000000009219407064  0.0000019787199499478  0.0000000000010017354  0.0000000318643973220
0.0000000000001868363  0.0000000002874013208  0.0000000000063005746  0.0000004753972949751  0.0000116417693857473 3
0.0000000000000000000  0.0000000000000000000  0.0000000000000000199  0.0000000077083173602  0.0000000012276302186
```

Fig. 6.2 Python out of a decomposed tensor

## 6.2 Convergence

During our testing, we also noticed that convergence is particularly an issue when one sets the tolerance on the difference in fit to four decimal digits (while using the IEEE VAST 2007 contest dataset). Figures 6.4 and 6.5 summarize the various tests we have with the maximum number of iterations set at 40 and 50 respectively. We see clearly that when the tolerance on the difference in fit is greater than 4 decimal digits, the software performs well. The PARAFAC of the tensor obtained from the VAST 2007 contest dataset takes less than 10 minutes. However, when the tolerance is less than 4 decimal digits, the software runs for over an hour in some instances. The choice of the tolerance is certainly problem-dependant. But in our tests, it was one the most interesting observation. That is why it is being brought to the attention of the user/reader.

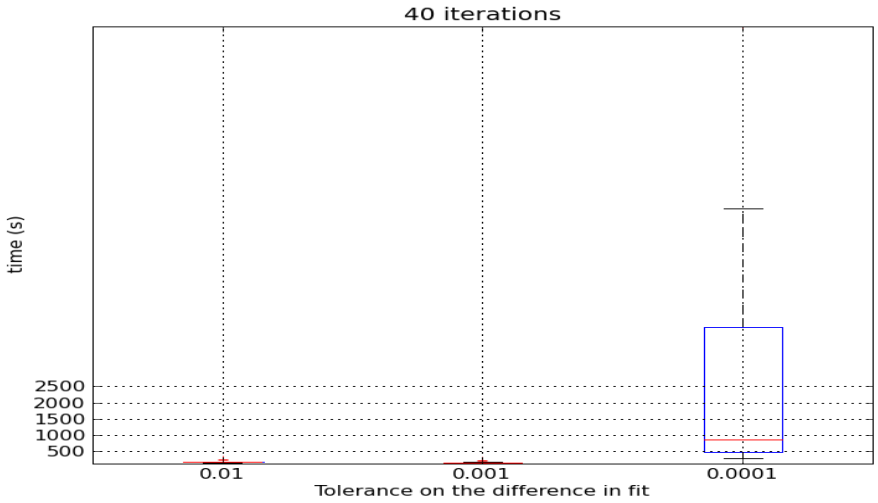


Fig. 6.3 Results of various runs with a maximum of 40 iterations

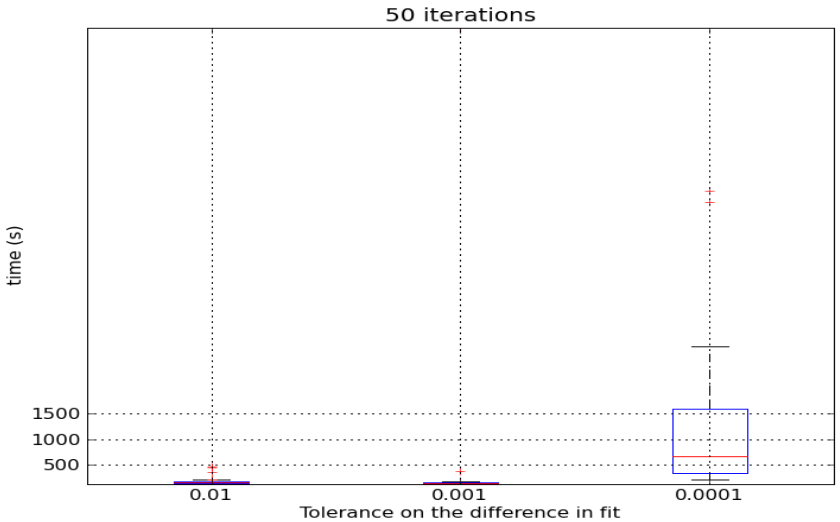


Fig. 6.4 Results of various runs with a maximum of 50 iterations

# 7 Conclusion

The goal of this project was to create a tool that would serve mainly two purposes. First, offer an alternative to the Tensor Toolbox for MATLAB. Second, provide a way to incorporate nonnegative factorization into FutureLens. At the end of this project, we believe that both goals are within reach. However, one needs to understand that NTF is only part of a bigger picture. Our software alone cannot lead to meaningful results. However when combined with Post -processing tools such as FutureLens and Expertise, it can lead to discovering meaningful hidden patterns large data sets. In an area such as Tensor decomposition, numerous additions can be made. An implementation of the Tucker model would be beneficial to the scientific community. Plus, the incorporation of NTF into FutureLens would greatly benefit text mining.

# 8 Acknowledgements

The author of this report would like to acknowledge Mr. Andrey Puretskiy for the discussions and helpful feedback at all stages of this project. Also, the Tensor Toolbox for MATLAB, by Bader and Kolda [1], contributed greatly to my understanding of tensor decomposition and the inner-workings of PARAFAC.

# 9 References

[1] Tamara G. Kolda, Brett W. Bader , "Tensor Toolbox For Matlab", http://csmr.ca.sandia.gov/~tgkolda/TensorToolbox/.

[2]

[3] Tamara G. Kolda, Brett W. Bader , "Tensor Decompostions and Applications", SIAM Review , June 10, 2008.

[4] Andrzej Cichocki, Rafal Zdunek, Anh Huy Phan, Shun-ichi Amari, "Nonnegative Matrix and Tensor Factorizations", John Wiley & Sons, Ltd, 1009.

[5] http://docs.python.org/library/profile.html, visited on 06/15/2010.

[6] http://www.mathworks.com/access/helpdesk/help/techdoc, visited on on 03/12/2010.

[7] http://www.scipy.org/NumPy_for_Matlab_Users, visited on 03/12/2010.

[8] Brett W. Bader, Andrey A. Puretskiy, Michael W. Berry, "Scenario Discovery Using Nonnegative Tensor Factorization", J. Ruiz-Schulcloper and W.G. Kropatsch (Eds.): CIARP 2008, LNCS 5197, pp.791-805, 2008.

[9] http://docs.scipy.org/doc/numpy/user/, visited on 03/14/2010.

[10] http://docs.scipy.org/doc/, visited on 03/14/2010.

[11] http://docs.scipy.org/doc/numpy/user/whatisnumpy.html, visited on 03/14/2010

[12] Tamara G. Kolda, "Multilinear operators for higher-order decompositions", SANDIA REPORT, April 2006.

[13] http://speleotrove.com/decimal/decifaq1.html#inexact, visited 04/10/2010.

[14] Hans Petter Langtangen, "A Primer on Scientific Programming with Python", Springer Dordrecht Heidelberg London New York, 2009.

[15] http://docs.python.org/library/profile.html, visited 05/15/2010.

[16] Hans Petter Langtangen, "Python Scripting for Computational Science", Springer, 3rd edition, 2009.

[17] http://www.mathworks.com/access/helpdesk/help/techdoc/ref/accumarray.html, visited 04/10/2010.

[18] Gregory Shutt, Andrey Puretskiy, Michael W. Berry, "FutureLens", Department of Electrical Engineering and Computer Science, The University Of Tennessee, November 20, 2008.

[19] http://docs.python.org/tutorial/floatingpoint.html, visited 03/25/2010.