

École polytechnique de Louvain

Block term tensor decomposition by numerical optimization

Author : Jonas DUBOIS

Supervisors : Francois GLINEUR, Guillaume OLIKIER

Readers : Pierre-Antoine ABSIL, Nico VERVLiet

Academic year 2018–2019

Master [120] in Mathematical Engineering

Contents

1	Introduction	1
2	Tensor algebra	5
2.1	Vectorization and matricization	5
2.2	Operations with tensors	6
2.2.1	Basic algebraic operations	6
2.2.2	Mode- N product	7
2.2.3	Outer Product	8
2.2.4	Mode- N product with unfold tensors	8
3	Tensor decompositions	9
3.1	CANDECOMP/PARAFAC	9
3.1.1	Tensor rank	10
3.2	Tucker decomposition	10
3.2.1	The n -Rank of a tensor	11
3.3	Block term decomposition	11
4	Computation of the block term decomposition	14
4.1	Formulation of the problem	14
4.1.1	Computation of the gradient	14
4.2	Method of resolution	18
4.3	Coordinate descent	18
4.4	The algorithms	19
4.4.1	Als , matrix by matrix	19
4.4.2	Als , vectors by vectors	20
4.4.3	Als , heuristics of the orders	22
4.5	Complexity	23
4.5.1	Theoretical complexity	23
4.5.2	Empirical measures	24
4.6	Optimization of the algorithms	25
4.7	State of the art	27
5	Numerical results	28
5.1	Tests	28
5.2	Tensorlab algorithms performance	29
5.3	Als performance	30
5.4	Influence of different parameters	32
5.4.1	Influence of the number of terms	32
5.4.2	Influence of the noise on the main tensor	33
5.4.3	Influence of the size of the main tensor	34
5.4.4	Influence of the size of the core tensors	35
5.4.5	Influence of the initial point	36
5.4.6	Conclusion on the influence of the parameters	37
5.5	<i>line</i> algorithms	38
5.5.1	CG influence	40
5.6	<i>steep</i> and <i>rand</i> algorithms	41
5.6.1	Influence of the orthogonality of the matrices	43

5.7	Final results	44
5.8	Multistarts	46
5.9	Interpretation of the results	47
6	Conclusion	49
7	Bibliography	50
8	Annexes	53
8.1	Matlab implementations	53
8.1.1	functions implemented in this thesis to solve a BTD	53
8.1.2	scripts implemented in this thesis to test the performance of the algorithms	53
8.1.3	Tensorlab functions and other used functions	54
8.2	Pseudo-codes	54

1 Introduction

Collecting data and, more importantly, learning information from these data has become a major issue in the world of technologies. All this information is mainly stored in the form of matrices, for which a dense field of the mathematics has been developed to analyze them, to solve them and, more importantly, to decompose them. Theory behind matrices helps solving completion problems, compressing data, creating models.

But one can ask what we would be able to produce if we used tensors instead to collect these data, i.e. if we used more than two dimensions to represent our data. The use of tensors in machine learning, signal processing and more, has grown in popularity over the last years.

The motivation in the use of tensor depends largely on the cases, but the general objective is to create more connections between the data, which would result in obtaining more information from the data set.

Thus, instead of decomposing a set of matrices independently, we can decompose the tensor obtained by stacking the different matrices; that way, we can learn more about the data set, if the matrices have some relation between them.

A simple way to create a third-order tensor from matrices would be by adding a time component to data, a set of two dimensional matrices would then become a three dimensional tensor.

We can mention some examples that motivates the use of tensors in different scientific fields. An easy first example is a video, if a picture is stored as a matrix of pixels, a video can be stored as a tensor composed, slide by slide, by the pictures matrices of each frame.

Using the same approach, consider a “Netflix” problem, where the ratings are stored in a matrix with the user and the movies as axes or dimensions. By adding the time at which the movie as been rated as a third axis, it could add a trending component in the recommendation system where it would be more proposed when it comes out or when it gains popularity for some reason.

Facial recognition has become a popular technology due to its use in many smart phones. It has been shown that 2D facial recognition has issues in various situations, such as different lighting or angle, and that those could be solved using 3D facial recognition. To model these 3D faces, the use of 3D tensor is required. [8] [16]

In signal processing, interest in the field of tensors has grown thanks to their capacity of representing high-order statistics (HOS). This field developed new algebraic tools for tensors in order to work with HOS. [3] [9]

In machine learning, the use of tensors has grown in popularity. They can, for example, be used to save space with deep tensorized networks or for long-term forecasting. [1]

In natural language processing, in the case of topic modelling, using latent-Dirichlet allocation algorithms, we can replace the matrices that represent the probabilities words/topics and topics/documents by a tensor representing the probabilities between each of these three components. This could incorporate higher order relationship in the data and allow to discover hidden topics that would not be possible with the use of matrices. [1]

In the case of pictures, if we model a picture as a matrix of pixels, we must assign a color to each pixel. A color is not generally represented by a scalar, but rather by a vector of three scalar, each representing a percentage of primary colors (blue, green and

red). The picture would then become a tensor with a size three in depth. Modelling photos this way could help image completion using the colors from the neighbors to recover the pixels [20]. Using this method, we would be able to model hyperspectral images by increasing the size of the third dimension, in other terms, using more than three colors.

However, to use these tensors, we must be able to extract the information they contain and to do so we must decompose them.

A decomposition is the transformation of a representation, in our case the representations are tensors, into an equivalent representation, generally formed by simpler components. This is why decompositions such as CANDECOMP/PARAFAC, the Tucker decomposition or, the one studied in this thesis, the block term decomposition, have been developed. To obtain these decompositions, methods must be implemented so that the obtained decomposition fits the tensor as accurately as possible. And this is what we will try to achieve in this thesis.

The BTM, specifically, has proven to be useful to model epileptic seizure by producing better results than the CP and the Tucker decomposition [12]. As the use of tensor grows, we could potentially discover many other pertinent use of the BTM.

Since The use of higher-order tensors and their decompositions is more complicated than the use of matrices. This is due to the increase in dimension that makes these problems more difficult, it is a case of the curse of dimensionality.

This complication is also due to the loss of basic tool of the linear algebra that does not translate exactly in the fields of multilinear algebra like matrix products or rank.

The objective of this thesis is to implement an algorithm that produce the best block terms decomposition of a tensor. The goal of our method is to match, or improve if possible, the results obtained by the already existing algorithms, with a different approach. Finally, we will mainly focus on third-order tensors as they already give a lot of insight about the gap of difficulty encountered between the decomposition of second-order and third-order tensors while still being representable visually.

Definition and notation

A *tensor* is a multidimensional array. More formally, an N -way or N -th order tensor is an element of the tensor product of N vector spaces, each of which has its own coordinate system. A first-order tensor is a vector, a second-order tensor is a matrix and tensors of order three or higher are called higher-order tensors. [14]

A third-order tensor has three indices, as seen in Figure 1

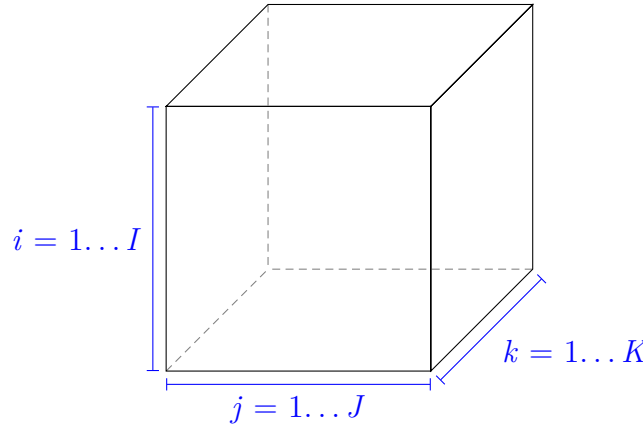


Figure 1: representation of a third order tensor

In this work, scalars are denoted by lowercase letters ($a, b, \dots$), vectors are denoted by boldface lowercase letters ($\mathbf{a}, \mathbf{b}, \dots$), matrices are written in boldface capital letters ($\mathbf{A}, \mathbf{B}, \dots$) and tensors are denoted by calligraphic letters ($\mathcal{A}, \mathcal{B}, \dots$).

Italic capital letters are used to denote sizes or index upper bounds, e.g. $n = 1, 2, \dots, N$. $\mathbb{R}$ and $\mathbb{N}$ represent respectively the sets of real and natural numbers, while the subscripts $\bullet_+$ and $\bullet_0$ denote the non-negative elements and non zero elements. Thence $\mathbb{R}_+^{(I \times J \times K)}$ is used to denote the set of size $(I \times J \times K)$ real non-negative three dimensional tensors, while $\mathbb{R}_+^{(I \times J)}$ denotes the set of size $(I \times J)$ real non-negative matrices.

This notation will be used to describe elements of a given structure, for instance, $(\mathcal{A})_{ijk} = a_{ijk}$ is the i -th index row, j th index column and k th index depth of the tensor $\mathcal{A}$.

To continue in that logic, the j th row vector of the matrix $\mathbf{A}$ is denoted $\mathbf{a}_j$, i.e.,

$$\mathbf{A} = [\mathbf{a}_1; \mathbf{a}_2; \dots].$$

$\mathbb{R}^{(N \times N)}$ denotes the set of square matrices of size N (equal number of elements on the rows and columns) and $\mathbb{R}^{(N \times N \times N)}$, the set of cubic tensors of size N (equal number of elements on the rows, columns and depths).

The norm used is the Frobenius norm, defined by $\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^I \sum_{j=1}^J |a_{ij}|^2}$, for $\mathbf{A} \in \mathbb{R}^{(I \times J)}$

matrices and $\|\mathcal{A}\|_F = \sqrt{\sum_{i=1}^I \sum_{j=1}^J \sum_{k=1}^K |a_{ijk}|^2}$, for $\mathcal{A} \in \mathbb{R}^{(I \times J \times K)}$ tensors.

Symbol $\otimes$ denotes the Kronecker product,

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots \\ a_{21}B & a_{22}B & \dots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

The superscripts $\bullet^\top$, $\bullet^{-1}$ and $\bullet^\dagger$ denotes the transpose, the inverse and the Moore-Penrose pseudo-inverse of a matrix, respectively.

A matrix $\mathbf{A} \in \mathbb{R}^{M \times N}$ is said to be orthogonal if

$$\mathbf{A}^\top \mathbf{A} = \mathbf{I}_N$$

where $\mathbf{I}_N$ denotes the identity matrix of size N .

2 Tensor algebra

This section will set the basis of tensor algebra. The goal is not to be exhaustive but rather to introduce all the concepts needed in the rest of the thesis.

2.1 Vectorization and matricization

The mode- n vectors, for $n = 1, \dots, N$, of a N -th order tensor are the generalization of rows and columns of a matrix for higher-order tensors. They are obtained by varying the n -th index and fixing all the other indices.

Mode- n vectors can also be called fibers, for third-order tensor, the mode-1 and mode-2 vectors will still be called rows and columns, respectively, like their matrix counterparts, while the mode-3 fibers will be called tubes. Fibers are represented in Figure 3.

A third-order tensor, is composed of “slices” of matrices.

Figure 2¹ represents in orange the mode-1 slice matrices, called horizontal slices, in green the mode-2 slice matrices, called lateral slices, and in blue the mode-3 slice matrices, called frontal slices.

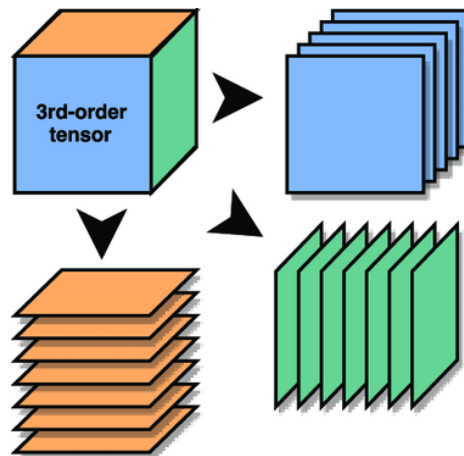


Figure 2: Graphic representation of slice matrix for a third-order tensor

Let $\mathcal{T} \in \mathbb{R}^{(I \times J \times K)}$ a third-order tensor composed by I horizontal slices, J lateral slices and K frontal slices.

Let $\mathbf{X}_i^n$, the i -th mode- n slice of $\mathcal{T}$ then

$$(\mathbf{X}_i^n)_{jk} = \mathcal{T}_{ijk}, \quad (\mathbf{X}_i^n)_{jk} = \mathcal{T}_{jik}, \quad (\mathbf{X}_i^n)_{jk} = \mathcal{T}_{jki}$$

if $n = 1, 2, 3$, respectively, for $j = 1, \dots, J$ and $k = 1, \dots, K$.

Manipulating higher-order tensors is relatively more complex than manipulating matrices. This is why it will often be useful to work with vectors or matrices representations of the tensors.

Vectorizing or matricizing a tensor consist in stacking, in the form of a vector or a matrix, the mode- n vectors of the tensor.

¹image from [24]

Let $\mathcal{T} \in \mathbb{R}^{(I_1 \times I_2 \times \dots \times I_N)}$, $\text{vec}(\mathcal{T})$ denotes the vectorization of $\mathcal{T}$, a vector of size $\prod_{n=1}^N I_n$ and is defined by:

$$(\text{vec}(\mathcal{T})) \left(\sum_{n=1}^{N-1} (i_n - 1) \prod_{m=n+1}^N I_m + i_N \right) := \mathcal{T}(i_1, \dots, i_N)$$

for each $(i_1, \dots, i_N) \in \prod_{n=1}^N \{1, \dots, I_N\}$.

Useful toolboxes, for MATLAB, to manipulate tensors in general are the ones provided by **Tensorlab**[26] and **Tensortoolbox**[2], in this thesis however, we will only use the Tensorlab toolbox.

It provides a library of functions that allows tensor manipulation, decompositions, modelling and more. The toolbox has been of a great help throughout this thesis.

In this case, **Tensorlab** provides a function that allows the vectorization of tensors, for example with a third-order tensor $\mathcal{T}$: **tens2vec**(**T**,[**3,2,1**]).

$\mathbf{T}_{(n)}$ for each $n \in \{1, \dots, N\}$ denotes the mode- n matricization of the tensor $\mathcal{T}$.

$$\mathbf{T}_{(n)} \left(i_n, 1 + \sum_{k=1}^{n-1} (i_k - 1) \prod_{l=k+1}^{n-1} I_l + \left(\prod_{m=1}^{n-1} I_m \right) \sum_{k=n+1}^N (i_k - 1) \prod_{l=k+1}^N I_l \right) := \mathcal{T}(i_1, \dots, i_N)$$

for each $(i_1, \dots, i_N) \in \prod_{n=1}^N \{1, \dots, I_N\}$, making $\mathbf{T}_{(n)} (I_n, \prod_{\substack{m=1 \\ m \neq n}}^N I_m)$ whose columns are the mode- n vectors of $\mathcal{T}$.

Tensorlab provides a function that allows the matricizations of tensors, for example with a third-order tensor $\mathcal{T}$ we obtain $\mathbf{T}_{(1)}$, $\mathbf{T}_{(2)}$, $\mathbf{T}_{(3)}$ by respectively: **tens2mat**(**T**,[**1,3,2**]), **tens2mat**(**T**,[**2,1,3**]) and **tens2mat**(**T**,[**3,2,1**]).

The transformation of tensors into vectors and matrices, (resp. the inverse transformation back into a tensor) can also be called unfolding (resp. folding) of a tensor.

One can see in Figure 3² that the first column represents the mode- n vectors, $n = 1, 2, 3$, and the second column represents of a matricization in each of the corresponding mode.

2.2 Operations with tensors

2.2.1 Basic algebraic operations

The sum between two tensors, $\mathcal{T}, \mathcal{B} \in \mathbb{R}^{I_1 \times \dots \times I_N}$ is defined elementwise and thus, is only defined for tensors with the same dimensions:

$$(\mathcal{A} + \mathcal{B})(i_1, \dots, i_N) := \mathcal{A}(i_1, \dots, i_N) + \mathcal{B}(i_1, \dots, i_N)$$

for each $(i_1, \dots, i_N) \in \prod_{n=1}^N \{1, \dots, I_n\}$.

The product between a scalar $a \in \mathbb{R}$ and a tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times \dots \times I_N}$ is defined elementwise.

²image from [15]

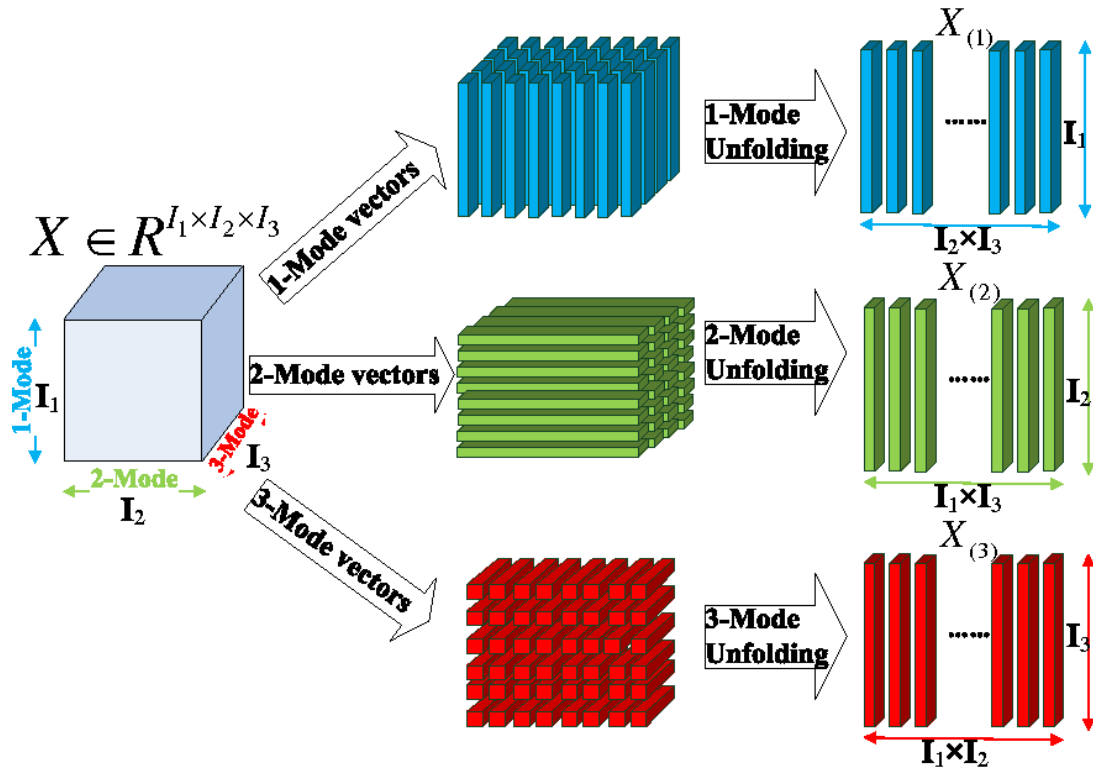


Figure 3: Graphic representation of mode- N vectors and matrix unfolding.

$$(a\mathcal{T})(i_1, \dots, i_N) := a\mathcal{T}(i_1, \dots, i_N)$$

for each $(i_1, \dots, i_N) \in \prod_{n=1}^N \{1, \dots, I_n\}$.

2.2.2 Mode- N product

To be able to work with tensors and matrices, we must first define a product between one and the other.

Let a tensor $\mathcal{T} \in \mathbb{R}^{I_1, \dots, I_N}$ and a matrix $\mathbf{U} \in \mathbb{R}^{J, I_n}$ for $n \in \{1, \dots, N\}$. The mode- N product is denoted by $\bullet_n$ $\forall n \in \{1, \dots, N\}$, and defined, elementwise, by:

$$(\mathcal{T} \bullet_n \mathbf{U})_{i_1, \dots, i_{n-1}, j, i_{n+1}, \dots, i_N} = \sum_{i_n=1}^{I_n} a_{i_1, \dots, i_N} u_{j, i_n}$$

for all indices.

This product can be thought of as the translation of the product between a vector and a matrix to higher-order tensors. For example, the difference between matrix that is multiply by a vector by its left or by its right, is the same as the one of a tensor multiply by a matrix by a mode-1 product or by a mode-2 product.

For a third-order tensor, let us consider $\mathcal{T} \in \mathbb{R}^{I \times J \times K}$, and $\mathbf{U} \in \mathbb{R}^{L \times I}$, $\mathbf{V} \in \mathbb{R}^{M \times J}$, $\mathbf{W} \in \mathbb{R}^{N \times K}$.

Then the products are defined by

Tucker mode-1 product: $(\mathcal{T} \bullet_1 \mathbf{U})_{ljk} = \sum_{i=1}^I t_{ijk} u_{li} \quad \forall l, j, k$

Tucker mode-2 product: $(\mathcal{T} \bullet_2 \mathbf{V})_{imk} = \sum_{j=1}^J t_{ijk} v_{mj} \quad \forall i, m, k$

Tucker mode-3 product: $(\mathcal{T} \bullet_3 \mathbf{W})_{ijn} = \sum_{k=1}^K t_{ijk} w_{nk} \quad \forall i, j, n$

2.2.3 Outer Product

The outer product $\mathcal{A} \circ \mathcal{B}$ of a tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_P}$ and $\mathcal{B} \in \mathbb{R}^{J_1 \times J_2 \times \dots \times J_Q}$ is the tensor defined by

$$(\mathcal{A} \circ \mathcal{B})_{i_1 \dots i_P j_1 \dots j_Q} = a_{i_1 \dots i_P} b_{j_1 \dots j_Q}$$

For example, the outer product of three vectors $\mathbf{a}, \mathbf{b}$ and $\mathbf{c}$ gives a tensor $\mathbf{a} \circ \mathbf{b} \circ \mathbf{c} = \mathcal{T}$ such that $t_{ijk} = a_i b_j c_k \quad \forall i, j, k$.

A tensor obtained by the outer product of three vectors is said to be rank-1, as seen in Figure 4.

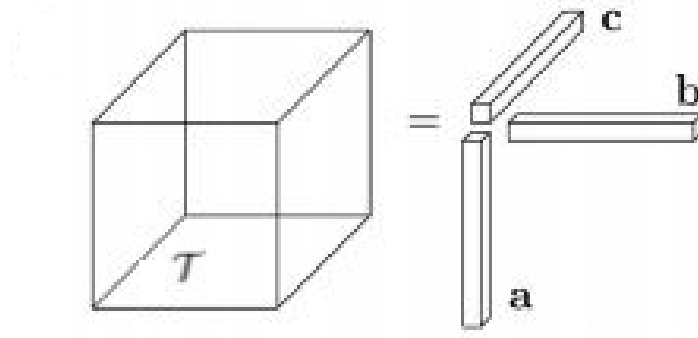


Figure 4: Rank-1 tensor obtained by the outer product of three vectors

2.2.4 Mode- N product with unfold tensors

The results of mode- N products can be vectorized or matricized thanks to the Kronecker product.

Let $\mathcal{T} \in \mathbb{R}^{I_1, \dots, I_N}$ and $\mathbf{U}_n \in \mathbb{R}^{I_n, J_n} \quad \forall n \in \{1, \dots, N\}$ then,

$$\text{vec}(\mathcal{T} \bullet_1 \mathbf{U}_1 \dots \bullet_N \mathbf{U}_N) = (\mathbf{U}_1 \otimes \dots \otimes \mathbf{U}_N) \text{vec}(\mathcal{T}) \quad (1)$$

If $\mathcal{P} := \mathcal{T} \bullet_1 \mathbf{U}_1 \dots \bullet_N \mathbf{U}_N$, then

$$\mathbf{P}_{(n)} = \mathbf{U}_n \otimes \mathbf{T}_{(n)} \left(\left(\bigotimes_{k=n+1}^N \mathbf{U}_k \right) \otimes \left(\bigotimes_{l=1}^{n-1} \mathbf{U}_l \right) \right)^T.$$

Which, for a third-order tensor $\mathcal{T}$, with $\mathbf{U} = \mathbf{U}_1, \mathbf{V} = \mathbf{U}_2, \mathbf{W} = \mathbf{U}_3$, gives the following formulas:

$$\mathbf{P}_{(1)} = \mathbf{U} \mathbf{T}_{(1)} (\mathbf{V} \otimes \mathbf{W})^T \quad (2)$$

$$\mathbf{P}_{(2)} = \mathbf{V} \mathbf{T}_{(2)} (\mathbf{W} \otimes \mathbf{U})^T \quad (3)$$

$$\mathbf{P}_{(3)} = \mathbf{W} \mathbf{T}_{(3)} (\mathbf{U} \otimes \mathbf{V})^T \quad (4)$$

These operations are important because they are the ones that form the Tucker and the block term decomposition.

3 Tensor decompositions

Now that we have introduced the algebraic background needed, we can move on to one of the aspects that justify their use, the tensor decompositions.

3.1 CANDECOMP/PARAFAC

The goal of the CANDECOMP/PARAFAC (canonical decomposition/parallel factors, CP) is to decompose a N -order tensor into a sum of rank-one tensors and thus by the sum of several outer products of N vectors.

Let a third-order tensor $\mathcal{T} \in \mathbb{R}^{I,J,K}$, the CP decomposition of $\mathcal{T}$ is defined by:

$$\mathcal{T} = \sum_{r=1}^R \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r$$

as represented in Figure 5, where $R \in \mathbb{R}_+$ is the number of terms, $\mathbf{a}_r \in \mathbb{R}^I$, $\mathbf{b}_r \in \mathbb{R}^J$ and $\mathbf{c}_r \in \mathbb{R}^K \quad \forall r \in \{1, \dots, R\}$.

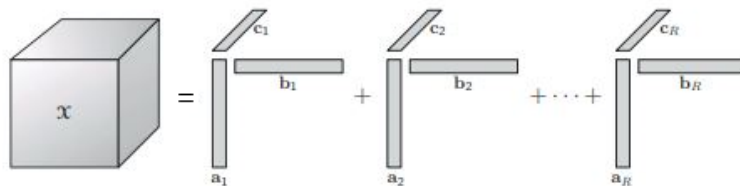


Figure 5: CANDECOMP/PARAFAC decomposition of a third-order tensor

Elementwise, we have:

$$t_{ijk} = \sum_{r=1}^R a_{ir} b_{jr} c_{kr}$$

for $i = 1, \dots, I$, $j = 1, \dots, J$ and $k = 1, \dots, K$.

CP has applications in many scientific fields. To name a few, it was proved that it had useful applications in chemometrics, in telecommunications, in independent component analysis (ICA) and in image compression and classification to cite only a few.[14]

3.1.1 Tensor rank

The *rank* of a tensor $\mathcal{T}$, denoted $\text{rank}(\mathcal{T})$, is defined as the smallest number of rank-1 tensors (see section 3.2.3) that generates $\mathcal{T}$ as their sum.

In other words, this is the smallest number of terms needed to obtain an exact CP decomposition.

Thus, $\text{rank}(\mathcal{T}) \leq R$ if $\mathcal{T} = \sum_{r=1}^R \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r$.

Since there is no theoretical method to find the rank of a tensor, the CP decomposition is the best way to know the rank of a tensor.

To do so, one must recover the exact original tensor with R terms of the decomposition, while not being able to do so with $R - 1$ terms.

3.2 Tucker decomposition

The Tucker decomposition is a form of higher-order SVD. It decomposes a tensor into a core tensor multiplied by a factor matrix along each mode.

Let a N -order tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times \dots \times I_N}$, $\mathcal{A} \in \mathbb{R}^{J_1 \times \dots \times J_N}$ and N matrices $\mathbf{U}_n \in \mathbb{R}^{I_n \times J_n} \quad \forall n \in \{1, \dots, N\}$, then

$$\mathcal{T} = \mathcal{A} \bullet_1 \mathbf{U}_1 \dots \bullet_N \mathbf{U}_N$$

Which, for a third-order tensor, gives

$$\mathcal{T} = \mathcal{A} \bullet_1 \mathbf{U} \bullet_2 \mathbf{V} \bullet_3 \mathbf{W}$$

where $\mathcal{T} \in \mathbb{R}^{I \times J \times K}$, $\mathcal{A} \in \mathbb{R}^{L \times M \times N}$, $\mathbf{U} \in \mathbb{R}^{I \times L}$, $\mathbf{V} \in \mathbb{R}^{J \times M}$ and $\mathbf{W} \in \mathbb{R}^{K \times N}$, as represented in Figure 6.

In this case, $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$ are the factor matrices and can be thought of as the principal component in each mode; they are usually orthogonal, by definition.

$\mathcal{A}$ is the core tensor, its entries can be seen as the level of interaction between the different components.

The matrices that multiply a tensor by a mode- n product will be called mode- n matrices for the rest of this paper.

Elementwise, the decomposition is

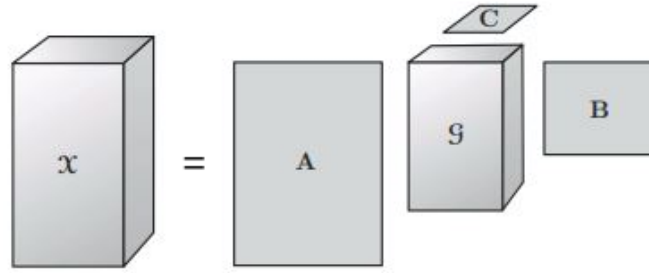


Figure 6: Tucker decomposition of a third-order tensor.

The factor matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ correspond to $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$, respectively, $\mathcal{G}$ correspond to $\mathcal{A}$ and $\mathcal{X}$ correspond to $\mathcal{T}$ in the notation of this thesis

$$t_{ijk} = \sum_{l=1}^L \sum_{m=1}^M \sum_{n=1}^N a_{lmn} u_{il} v_{jm} w_{kn} \quad \text{for } i = 1, \dots, I, j = 1, \dots, J, k = 1, \dots, K$$

If L, M, N are smaller than I, J, K , the core tensor $\mathcal{A}$ can be thought of as a compression of the tensor $\mathcal{T}$.

The Tucker decomposition has seen applications in area such as signal processing, in chemical analysis and in computer vision with the TensorFaces to only cite a few.[14]

3.2.1 The n -Rank of a tensor

Let $\mathcal{T}$ be an N -th order tensor of size $I_1 \times I_2 \times \dots \times I_N$. Then the n -rank, also called multilinear rank, of $\mathcal{T}$, denoted $\text{rank}_n(\mathcal{T})$, is the column rank of $\mathbf{T}_{(n)}$. In other words, the n -rank is the dimension of the vector space spanned by the mode- n vectors. If we let $R_n = \text{rank}_n(\mathcal{T})$ for $n = 1, \dots, N$, then we can say that $\mathcal{T}$ is a rank- $(R_1, R_2, \dots, R_N)$ tensor. For a given tensor $\mathcal{T}$, we can easily find an exact Tucker decomposition of rank $(R_1, R_2, \dots, R_N)$, where $R_n = \text{rank}_n(\mathcal{T})$, because the core tensor and the main tensor $\mathcal{T}$ would have the same n -Rank. If, however, we compute a Tucker decomposition of rank $(R_1, R_2, \dots, R_N)$, where $R_n < \text{rank}_n(\mathcal{T})$ for at least one n , then it will be necessarily an inexact decomposition and more difficult to compute.

A decomposition is said to be exact when the tensor formed by this decomposition is perfectly equal to the main tensor that we decomposed, thus, an inexact decomposition form a tensor that approximates the main tensor, but is not equal to it.

3.3 Block term decomposition

The block term decomposition (BTD) is the combination of the CP decomposition and the Tucker decomposition, as the tensor is decomposed in a sum of lower multilinear rank tensors.

Let $\mathcal{T} \in \mathbb{R}^{I_1 \times \dots \times I_N}$, $\mathcal{A}^r \in \mathbb{R}^{J_1^r \times \dots \times J_N^r}$ and $\mathbf{U}_n^r \in \mathbb{R}^{I_n \times J_n^r} \quad \forall n \in \{1, \dots, N\}, \forall r \in \{1, \dots, R\}$, full column rank with $J_n^r \leq I_n$.

Then, a BTD of $\mathcal{T}$ is defined by

$$\mathcal{T} = \sum_{r=1}^R \mathcal{A}^r \bullet_1 \mathbf{U}_1^r \dots \bullet_N \mathbf{U}_N^r$$

Let $\mathcal{T} \in \mathbb{R}^{I \times J \times K}$, $\mathcal{A}_r^{L_r \times M_r \times N_r}$ and $\mathbf{U}_r^{I \times L_r}$, $\mathbf{V}_r^{J \times M_r}$, $\mathbf{W}_r^{K \times N_r} \forall r \in \{1, \dots, R\}$, this gives for third-order tensors:

$$\mathcal{T} = \sum_{r=1}^R \mathcal{A}^r \bullet_1 \mathbf{U}_r \bullet_2 \mathbf{V}_r \bullet_3 \mathbf{W}_r, \quad (5)$$

represented in Figure 7³.

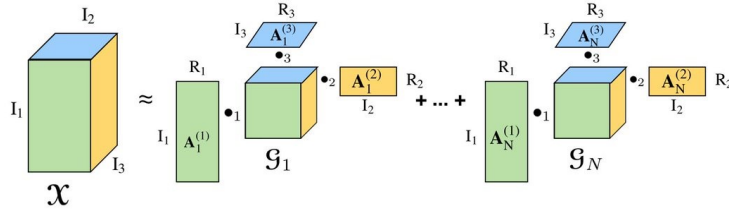


Figure 7: block term decomposition of a third-order tensor in N terms. The factor matrices $\mathbf{A}_n^{(1)}$, $\mathbf{A}_n^{(2)}$ and $\mathbf{A}_n^{(3)}$ correspond to $\mathbf{U}_r$, $\mathbf{V}_r$ and $\mathbf{W}_r$, respectively, $\mathcal{G}_n$ correspond to $\mathcal{A}_r$ and $\mathcal{X}$ correspond to $\mathcal{T}$ in the notation of this thesis

The unfolded equations are given by:

$$\mathbf{T}_{(1)} = \sum_{r=1}^R \mathbf{U}_r (\mathbf{A}_r)_{(1)} (\mathbf{V}_r \otimes \mathbf{W}_r)^T \quad (6)$$

$$\mathbf{T}_{(2)} = \sum_{r=1}^R \mathbf{V}_r (\mathbf{A}_r)_{(2)} (\mathbf{W}_r \otimes \mathbf{U}_r)^T \quad (7)$$

$$\mathbf{T}_{(3)} = \sum_{r=1}^R \mathbf{W}_r (\mathbf{A}_r)_{(3)} (\mathbf{U}_r \otimes \mathbf{V}_r)^T \quad (8)$$

The sum of two tensors being elementwise, the operation is commutative, one can then arbitrarily permute different terms of Equation (5). Also, one can multiply $\mathbf{U}_r$ by a non-singular matrix $\mathbf{P}_r \in \mathbb{R}^{L_r \times L_r}$, $\mathbf{V}_r$ by a non-singular matrix $\mathbf{Q}_r \in \mathbb{R}^{M_r \times M_r}$, $\mathbf{W}_r$ by a non-singular matrix $\mathbf{R}_r \in \mathbb{R}^{N_r \times N_r}$, if $\mathcal{A}_r$ is replaced by $\mathcal{A}_r \bullet_1 \mathbf{P}_r^{-1} \bullet_2 \mathbf{Q}_r^{-1} \bullet_3 \mathbf{R}_r^{-1}$. A square matrix is said non-singular, i.e., one that has an inverse matrix, if and only if its determinant is nonzero.

De Lathauwer has demonstrated the theorem of essential uniqueness of the BTD [6]. A decomposition is said to be essentially unique when it is subject only to these trivial indeterminacies.

Equation 5 is essentially unique if:

- $L_r = L$, $M_r = M$ for $r = 1, \dots, R$
- $L = M$

³image from [27]

- $I \geq LR$ and $J \geq MR$
- $N_r = N \geq 3$ and $\mathbf{W}_r$ is full column rank for each $r \in \{1, \dots, R\}$.

This is the decomposition that will be studied for the rest of this thesis.

4 Computation of the block term decomposition

In the previous section we introduced the background on higher-order tensors needed for this thesis. We also defined the block term decomposition. This section will present the algorithms implemented to fit a tensor by a BTM in a least square sense, while the next section will present the results obtained with these methods

This section, along with the next one, are the core of this project because they represent the main contribution of this work.

4.1 Formulation of the problem

To compute the BTM, we have to find a decomposition that fits the best the original tensor that we want to decompose.

Let $\mathcal{T} \in \mathbb{R}^{I \times J \times K}$, for $I, J, K \in \mathbb{N}_0$, the goal of the problem is to find core tensors $\mathcal{A}_r \in \mathbb{R}^{L_r \times M_r \times N_r}$ and factor matrices $\mathbf{U}_r \in \mathbb{R}^{I \times L_r}$, $\mathbf{V}_r \in \mathbb{R}^{J \times M_r}$ and $\mathbf{W}_r \in \mathbb{R}^{K \times N_r}$, for $r = 1, \dots, R$, and for $R \in \mathbb{N}_0$ that minimizes the cost function. The number of terms R and the sizes of the core tensors L_r, M_r, N_r are a priori given.

The objective function is defined such as:

$$f_{\mathcal{T}} : \prod_{r=1}^R (\mathbb{R}^{L_r \times M_r \times N_r}) \times \prod_{r=1}^R (\mathbb{R}^{I \times L_r}) \times \prod_{r=1}^R (\mathbb{R}^{J \times M_r}) \times \prod_{r=1}^R (\mathbb{R}^{K \times N_r}) \rightarrow \mathbb{R} :$$

$$(\mathcal{A}_1, \dots, \mathcal{A}_R, \mathbf{U}_1, \dots, \mathbf{U}_R, \mathbf{V}_1, \dots, \mathbf{V}_R) \mapsto \left\| \mathcal{T} - \sum_{r=1}^R \mathcal{A}_r \bullet_1 \mathbf{U}_r \bullet_2 \mathbf{V}_r \bullet_3 \mathbf{W}_r \right\|_F^2 \quad (9)$$

To do so, we have to minimize the error between the tensor and its decomposition. We will use the the least square error as cost function, as seen in Equation 9.

The objective is to minimize the cost function, i.e. to solve the following optimization problem:

$$\underset{\mathcal{A}_r, \mathbf{U}_r, \mathbf{V}_r, \mathbf{W}_r}{\text{minimize}} \quad f_{\mathcal{T}}(\mathcal{A}_1, \dots, \mathcal{A}_R, \mathbf{U}_1, \dots, \mathbf{U}_R, \mathbf{V}_1, \dots, \mathbf{V}_R, \mathbf{W}_1, \dots, \mathbf{W}_R)$$

The solving strategy of this problem in this project will be to apply an alternating least square (ALS) method and, hopefully, to converge towards the solution group of coordinates by group of coordinates.

To do so, we must first compute the gradient of the cost function for each of these groups of coordinates and equals them to null matrices or null tensors. We can then solve the system for the block of coordinate to minimize the objective function in the corresponding coordinate.

Note that (...) will replace $(\mathcal{A}_1, \dots, \mathcal{A}_R, \mathbf{U}_1, \dots, \mathbf{U}_R, \mathbf{V}_1, \dots, \mathbf{V}_R, \mathbf{W}_1, \dots, \mathbf{W}_R)$ as the argument of $f_{\mathcal{T}}$ in the rest of the section.

4.1.1 Computation of the gradient

Let

$$h : \prod_{r=1}^R (\mathbb{R}^{L_r \times M_r \times N_r}) \times \prod_{r=1}^R (\mathbb{R}^{I \times L_r}) \times \prod_{r=1}^R (\mathbb{R}^{J \times M_r}) \times \prod_{r=1}^R (\mathbb{R}^{K \times N_r}) \rightarrow \mathbb{R}^{I \times J \times K}$$

and

$$g : \mathbb{R}^{I \times J \times K} \rightarrow \mathbb{R}.$$

So that

$$f_{\mathcal{T}}(\dots) = (g \circ h)_{\mathcal{T}}(\dots).$$

Where $g(x) = \|x\|_F^2$ and $h(\dots) = \mathcal{T} - \sum_{r=1}^R \mathcal{A}_r \bullet_1 \mathbf{U}_r \bullet_2 \mathbf{V}_r \bullet_3 \mathbf{W}_r$.

To compute the gradient, we must use the chain rule of the derivative.

$$\frac{\partial}{\partial x} (g \circ h)_{\mathcal{T}}(\dots) = \frac{\partial g(h(x))}{\partial x} \frac{\partial h(x)}{\partial x}^{\top},$$

for $x = (\mathcal{A}_1, \dots, \mathcal{A}_R, \mathbf{U}_1, \dots, \mathbf{U}_R, \mathbf{V}_1, \dots, \mathbf{V}_R, \mathbf{W}_1, \dots, \mathbf{W}_R)$, any element of the BTD, i.e. a factor matrix or a core tensor.

First, we can compute the gradient of the squared norm, which is given by:

$$\frac{\partial g(x)}{\partial x} = \frac{\partial \|x\|_F^2}{\partial x} = 2x$$

Second, to compute $\frac{\partial h(x)}{\partial x}$, we can use Equations 1, 2, 3 and 4, so that we have;

$$\begin{aligned} h(\dots) &= \text{vec}(\mathcal{T}) - \sum_{s=1}^R (\mathbf{U}_r \otimes \mathbf{V}_r \otimes \mathbf{W}_r) \text{vec}(\mathcal{A}_r) \\ &= \mathbf{T}_{(1)} - \sum_{s=1}^R \mathbf{U}_r (\mathbf{A}_r)_{(1)} (\mathbf{V}_r \otimes \mathbf{W}_r)^{\top} \\ &= \mathbf{T}_{(2)} - \sum_{s=1}^R \mathbf{V}_r (\mathbf{A}_r)_{(2)} (\mathbf{W}_r \otimes \mathbf{U}_r)^{\top} \\ &= \mathbf{T}_{(3)} - \sum_{s=1}^R \mathbf{W}_r (\mathbf{A}_r)_{(3)} (\mathbf{U}_r \otimes \mathbf{V}_r)^{\top}. \end{aligned}$$

We can now proceed to compute the derivatives like the derivative of a matrix product [22]. We obtain:

$$\begin{aligned} \frac{\partial h(\dots)}{\partial \mathcal{A}_r} &= (\mathbf{U}_r \otimes \mathbf{V}_r \otimes \mathbf{W}_r), \\ \frac{\partial h(\dots)}{\partial \mathbf{U}_r} &= (\mathbf{A}_r)_{(1)} (\mathbf{V}_r \otimes \mathbf{W}_r)^{\top}, \\ \frac{\partial h(\dots)}{\partial \mathbf{V}_r} &= (\mathbf{A}_r)_{(2)} (\mathbf{W}_r \otimes \mathbf{U}_r)^{\top}, \\ \frac{\partial h(\dots)}{\partial \mathbf{W}_r} &= (\mathbf{A}_r)_{(3)} (\mathbf{U}_r \otimes \mathbf{V}_r)^{\top}. \end{aligned}$$

For $r = 1, \dots, R$

Now, we can compute the gradient of the cost function $f_{\mathcal{T}}$.

Gradient with respect to the core tensors

Using the vectorized equation, the gradient is given by:

$$\begin{aligned}\text{vec}(\nabla_{\mathcal{A}_r} f_{\mathcal{T}})(\dots) &= 2(\mathbf{U}_r \otimes \mathbf{V}_r \otimes \mathbf{W}_r)^\top \left(\sum_{s=1}^R (\mathbf{U}_s \otimes \mathbf{V}_s \otimes \mathbf{W}_s) \text{vec}(\mathcal{A}_s) - \text{vec}(\mathcal{T}) \right) \\ &= 2 \left(\sum_{s=1}^R (\mathbf{U}_r^\top \mathbf{U}_s \otimes \mathbf{V}_r^\top \mathbf{V}_s \otimes \mathbf{W}_r^\top \mathbf{W}_s) \text{vec}(\mathcal{A}_s) - (\mathbf{U}_r^\top \otimes \mathbf{V}_r^\top \otimes \mathbf{W}_r^\top) \text{vec}(\mathcal{T}) \right)\end{aligned}$$

Because $(\mathbf{A} \otimes \mathbf{B})^\top = \mathbf{A}^\top \otimes \mathbf{B}^\top$.

Which, once folded back into a tensor form, gives:

$$\begin{aligned}(\nabla_{\mathcal{A}_r} f_{\mathcal{T}})(\dots) &= 2 \left(\sum_{s=1}^R \mathcal{A}_s \bullet_1 \mathbf{U}_s \bullet_2 \mathbf{V}_s \bullet_3 \mathbf{W}_s - \mathcal{T} \right) \bullet_1 \mathbf{U}_r^\top \bullet_2 \mathbf{V}_r^\top \bullet_3 \mathbf{W}_r^\top \\ &= 2 \left(\sum_{s=1}^R \mathcal{A}_s \bullet_1 (\mathbf{U}_r^\top \mathbf{U}_s) \bullet_2 (\mathbf{V}_r^\top \mathbf{V}_s) \bullet_3 (\mathbf{W}_r^\top \mathbf{W}_s) - \mathcal{T} \bullet_1 \mathbf{U}_r^\top \bullet_2 \mathbf{V}_r^\top \bullet_3 \mathbf{W}_r^\top \right)\end{aligned}$$

for $r = 1, \dots, R$

Gradient with respect of the factor matrices

Using the matricized equations obtained previously, the gradients are given by:

$$\begin{aligned}(\nabla_{\mathbf{U}_r} f_{\mathcal{T}})(\dots) &= 2 \left(\sum_{s=1}^R \mathbf{U}_s (\mathbf{A}_s)_{(1)} (\mathbf{V}_s \otimes \mathbf{W}_s)^\top - \mathbf{T}_{(1)} \right) ((\mathbf{A}_r)_{(1)} (\mathbf{V}_r \otimes \mathbf{W}_r)^\top)^\top \\ &= 2 \left(\sum_{s=1}^R \mathbf{U}_r (\mathbf{A}_s)_{(1)} (\mathbf{V}_s^\top \mathbf{V}_r \otimes \mathbf{W}_r^\top \mathbf{W}_s) - \mathbf{T}_{(1)} (\mathbf{V}_r \otimes \mathbf{W}_r) (\mathbf{A}_r)_{(1)}^\top \right)\end{aligned}$$

$$\begin{aligned}(\nabla_{\mathbf{V}_r} f_{\mathcal{T}})(\dots) &= 2 \left(\sum_{s=1}^R \mathbf{V}_s (\mathbf{A}_s)_{(2)} (\mathbf{W}_s \otimes \mathbf{U}_s)^\top - \mathbf{T}_{(2)} \right) ((\mathbf{A}_r)_{(2)} (\mathbf{W}_r \otimes \mathbf{U}_r)^\top)^\top \\ &= 2 \left(\sum_{s=1}^R \mathbf{V}_r (\mathbf{A}_s)_{(2)} (\mathbf{W}_s^\top \mathbf{W}_r \otimes \mathbf{U}_r^\top \mathbf{U}_s) - \mathbf{T}_{(2)} (\mathbf{W}_r \otimes \mathbf{U}_r) (\mathbf{A}_r)_{(2)}^\top \right)\end{aligned}$$

$$\begin{aligned}(\nabla_{\mathbf{W}_r} f_{\mathcal{T}})(\dots) &= 2 \left(\sum_{s=1}^R \mathbf{W}_s (\mathbf{A}_s)_{(3)} (\mathbf{U}_s \otimes \mathbf{V}_s)^\top - \mathbf{T}_{(3)} \right) ((\mathbf{A}_r)_{(3)} (\mathbf{U}_r \otimes \mathbf{V}_r)^\top)^\top \\ &= 2 \left(\sum_{s=1}^R \mathbf{W}_r (\mathbf{A}_s)_{(3)} (\mathbf{U}_s^\top \mathbf{U}_r \otimes \mathbf{V}_s^\top \mathbf{V}_r) - \mathbf{T}_{(3)} (\mathbf{U}_r \otimes \mathbf{V}_r) (\mathbf{A}_r)_{(3)}^\top \right)\end{aligned}$$

for $r = 1, \dots, R$.

Now that we have the gradient for each of the groups of coordinate, we can solve the systems to optimize the cost function in each of its components. In other words, we must find $\mathbf{U}_r$, $\mathbf{V}_r$, $\mathbf{W}_r$ and $\mathcal{A}_r$ such that we have, respectively,

$$(\nabla_{\mathbf{U}_r} f_{\mathcal{T}})(\dots) = \mathbf{0}_{I, L_r},$$

$$(\nabla_{\mathbf{V}_r} f_{\mathcal{T}})(\dots) = \mathbf{0}_{J, M_r},$$

$$(\nabla_{\mathbf{W}_r} f_{\mathcal{T}})(\dots) = \mathbf{0}_{K, N_r},$$

$$(\nabla_{\mathcal{A}_r} f_{\mathcal{T}})(\dots) = \mathbf{0}_{L_r, M_r, N_r},$$

for $r = 1, \dots, R$.

The resolution of these systems is the main reason we computed the gradients, the objective function is quadratic for each group of coordinate so that the solution of the systems are minima.

Their solutions are given by:

equations for $\mathbf{U}_r$

$$\mathbf{U}_r = \left(\mathbf{T}_{(1)} - \sum_{\substack{s=1 \\ s \neq r}}^R \mathbf{U}_s (\mathbf{A}_s)_{(1)} (\mathbf{V}_s \otimes \mathbf{W}_s)^\top \right) ((\mathbf{A}_r)_{(1)} (\mathbf{V}_r \otimes \mathbf{W}_r)^\top)^\dagger \quad (10)$$

equations for $\mathbf{V}_r$

$$\mathbf{V}_r = \left(\mathbf{T}_{(2)} - \sum_{\substack{s=1 \\ s \neq r}}^R \mathbf{V}_s (\mathbf{A}_s)_{(2)} (\mathbf{W}_s \otimes \mathbf{V}_s)^\top \right) ((\mathbf{A}_r)_{(2)} (\mathbf{W}_r \otimes \mathbf{V}_r)^\top)^\dagger \quad (11)$$

equations for $\mathbf{W}_r$

$$\mathbf{W}_r = \left(\mathbf{T}_{(3)} - \sum_{\substack{s=1 \\ s \neq r}}^R \mathbf{W}_s (\mathbf{A}_s)_{(3)} (\mathbf{U}_s \otimes \mathbf{V}_s)^\top \right) ((\mathbf{A}_r)_{(3)} (\mathbf{U}_r \otimes \mathbf{V}_r)^\top)^\dagger \quad (12)$$

equations for $\mathcal{A}_r$

$$\begin{aligned} \mathcal{A}_r = & \left(\mathcal{T} \bullet_1 \mathbf{U}_r^\top \bullet_2 \mathbf{V}_r^\top \bullet_3 \mathbf{W}_r^\top - \sum_{\substack{s=1 \\ s \neq r}}^R \mathcal{A}_s \bullet_1 (\mathbf{U}_r^\top \mathbf{U}_s) \bullet_2 (\mathbf{V}_r^\top \mathbf{V}_s) \bullet_3 (\mathbf{W}_r^\top \mathbf{W}_s) \right) \\ & \bullet_1 (\mathbf{U}_r^\top \mathbf{U}_r)^{-1} \bullet_2 (\mathbf{V}_r^\top \mathbf{V}_r)^{-1} \bullet_3 (\mathbf{W}_r^\top \mathbf{W}_r)^{-1} \end{aligned} \quad (13)$$

for $r = 1, \dots, R$.

And because the factor matrices are orthogonal, we have:

$$\begin{aligned} \mathcal{A}_r = & \left(\mathcal{T} \bullet_1 \mathbf{U}_r^\top \bullet_2 \mathbf{V}_r^\top \bullet_3 \mathbf{W}_r^\top - \sum_{\substack{s=1 \\ s \neq r}}^R \mathcal{A}_s \bullet_1 (\mathbf{U}_r^\top \mathbf{U}_s) \bullet_2 (\mathbf{V}_r^\top \mathbf{V}_s) \bullet_3 (\mathbf{W}_r^\top \mathbf{W}_s) \right) \bullet_1 \mathbf{I}_{L_r} \bullet_2 \mathbf{I}_{M_r} \bullet_3 \mathbf{I}_{N_r} \\ = & \mathcal{T} \bullet_1 \mathbf{U}_r^\top \bullet_2 \mathbf{V}_r^\top \bullet_3 \mathbf{W}_r^\top - \sum_{\substack{s=1 \\ s \neq r}}^R \mathcal{A}_s \bullet_1 (\mathbf{U}_r^\top \mathbf{U}_s) \bullet_2 (\mathbf{V}_r^\top \mathbf{V}_s) \bullet_3 (\mathbf{W}_r^\top \mathbf{W}_s) \end{aligned}$$

4.2 Method of resolution

The computation of the best approximate BTB is a non-convex optimization problem [21] and difficult to realize, in part due to the number of elements in the decomposition. The elements in question being the three factor matrices and the core tensors of each term, there are then $4R$ elements in the decomposition, R being the number of terms. A way to solve this problem is to break it down to smaller, and thus, easier problems. This approach can be seen as a *divide and conquer* strategy for the decomposition of a tensor in block terms.

Each of the elements can be thought of as a coordinates (or groups of coordinates) of the cost function. Therefore, if we solve the problem element by element, the method becomes a coordinate descent algorithm.

4.3 Coordinate descent

Coordinate descent is an optimization technique that successively minimizes along coordinate directions, or groups of coordinate direction, i.e. blocks, to find the minimum of a function. At the start of each iteration, the algorithm determines an order for the coordinates or a blocks of coordinates via a coordinate selection rule. Then it, exactly or inexactly, minimizes the cost function over the corresponding coordinate direction while fixing all other coordinates or blocks of coordinates, one after an other, following the order fixed at the start of the iteration. Finally, once all the coordinates or groups of coordinates have been minimized, the algorithm proceeds to the next iteration.[4] [11]

The method is based on the idea that the minimization of a function $F(x_1, \dots, x_N)$ with N variables can be achieved by minimizing N optimization problems with only one variable in a loop. In the simplest case, one iterates cyclically through one direction at a time, minimizing the objective function with respect to that coordinate direction at the iteration.

Starting at an initial point $(x_1^0, \dots, x_N^0)$, one computes x_i^{k+1} at the iteration $k+1$ by solving the single variable optimization problems:

$$x_i^{k+1} = \underset{y}{\operatorname{argmin}}(x_1^{k+1}, \dots, x_{i-1}^{k+1}, y, x_{i+1}^k, \dots, x_N^k),$$

for $i = 1, \dots, N$.

We can see the representation of a simple case of coordinate descent in Figure 8 ⁴, with two variables (x, y) and the function $f(x, y) = 5x^2 - 6xy + 5y^2$ where the initial point is $(x_0, y_0) = (-0.5, -1.0)$. The solution of this optimization problem is trivial and is equal to $(0, 0)$, as shown.

In the case of the BTB, the coordinates or groups of coordinates are the core tensors and the factor matrices, and the objective function is the squared Frobenius norm of the error.

⁴example and image from: https://en.wikipedia.org/wiki/Coordinate_descent [4]

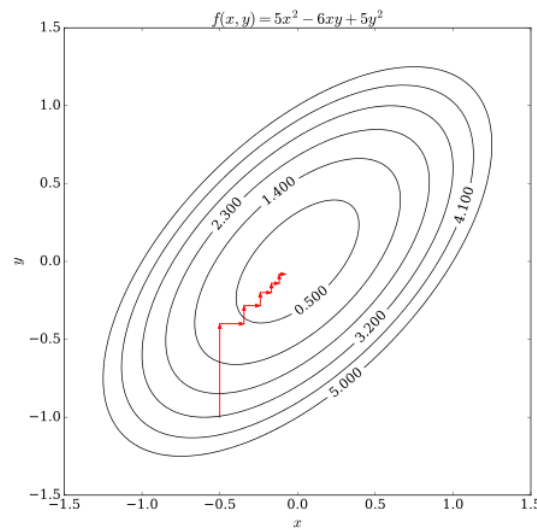


Figure 8: Exemple of coordinate descent

4.4 The algorithms

The algorithms implemented in this thesis are based on the concept of coordinate descent by alternating least square (*Als*) problem at each iteration.

The different methods keep iterating until they reach one of its stopping criteria:

- A maximum number of iteration has been reached.
- The norm of the gradient has fallen under the set tolerance.
- The relative error of the cost function, that will be defined later, has fallen under the set precision.

The pseudo-code of the different algorithms implemented in this thesis can be found in annexes

4.4.1 *Als*, matrix by matrix

The first method, called *Als*, considers the matrices as group of coordinates, thus alternate matrices by matrices. The order of updates is fixed, arbitrary and follows this order:

- first, all the matrices from the first dimension ($A_n^{(1)}$) in green,
- second, all the matrices from the second dimension ($A_n^{(2)}$) in yellow,
- third, all the matrices from the third dimension ($A_n^{(3)}$) in blue.
The matrices are then orthogonalized by a QR factorisation.
- Finally the core tensors $\mathcal{G}_n$ are updated,
for $n = 1, \dots, N$.

The description of the algorithm follows the notations and the colors of Figure 7. To update the matrices, one must solve Equations 10,11,12 and Equation 13 to update the core tensor.

Algorithm 1 represent the pseudo-code of *Als*.

The pseudo-code of the algorithm implemented in this thesis are available in annexes.

Note that the pseudo-code are using the notations of MATLAB.

4.4.2 *Als*, vectors by vectors

The second class of methods works in the same way but breaks down the problem in smaller groups of coordinates. The factor matrices are split into its mode-2 vectors, so each matrix will be I_n vectors of length J_n^r stacked onto each others, for I_n being the size of the tensor to decompose in the n -th mode and J_n^r being the size of the r -th core tensor in the n -th mode.

The goal is to solve for groups of coordinates as small as possible, for these methods, we solve systems to obtain vectors that compose the factor matrices, because each vector will only influence one slice matrix that composes the main tensor.

For example, when the i -th vector of the mode- n factor matrix is updated, only the mode- n slice matrix is modified, and this is true for all the terms of the decomposition.

The problem being more divided, we have more elements to update and thus, the possibility to do more permutation in the order during each iterations, the difference between each of the methods will be the order in which we iterate. This is why there will be three methods for this class of algorithms, which we call *line1*, *line2* and *line3*.

Let $\mathcal{T} \in \mathbb{R}^{(I \times J \times K)}$ be the main tensor, $\mathcal{S}_r \in \mathbb{R}^{(L_r \times M_r \times N_r)}$ and $\mathbf{U}_r \in \mathbb{R}^{(I \times L_r)}$, $\mathbf{V}_r \in \mathbb{R}^{(J \times M_r)}$, $\mathbf{W}_r \in \mathbb{R}^{(K \times N_r)}$ for $r = 1, \dots, R$, to demonstrate the difference between the methods.

- *line1*

The *line1* method will concentrate on each elements of the mode-1 factor matrices, the first line of each matrix, then the second line and so on until the I -th line.

Then repeat the process with the J lines of the R mode-2 matrices and proceed the K lines of the mode-3 matrices.

Finally, all the core tensors are updated.

The order of method *line1* is shown in Figure 9.

Algorithm 2 represent the pseudo-code of *line1*.

- *line2*

The *line2* method will first update on the first vectors of the mode-1 factor matrices, then the first vectors of the mode-2 and after the first line of the mode-3 matrices before going to the second vectors of the factor matrices in the same order. Then keep doing this loop N times, N being the largest dimension of the main tensor (I , J or K) and thus,

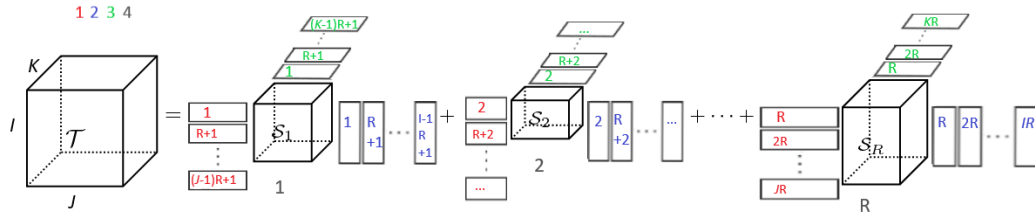


Figure 9: Order of update of method line 1

the maximum number of lines of the factor matrices.

Finally, all the core tensors are updated.

The order of method *line2* is shown in Figure 10.

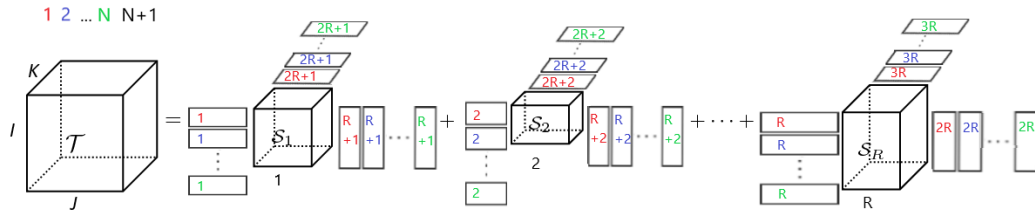


Figure 10: Order of update of method line 2

Notes: if $\mathcal{T}$ is not cubic, then $N > I$ or $N > J$ or $N > K$. In these cases, the iterations that go through the corresponding smaller dimension(s) stop at, respectively, I or J or K .

This is also valid for the next method.

Algorithm 3 represent the pseudo-code of *line2*.

- *line3*

The *line3* method will first update the first vectors of all factor matrices of the first term, then the first vectors of the factor matrices of the second term and keep on until the R -th term. Then proceed to repeat with the second vectors and so on until the N -th vectors of each factor matrix is updated.

Finally, all the core tensors are updated.

The order of method *line3* is shown in Figure 11.

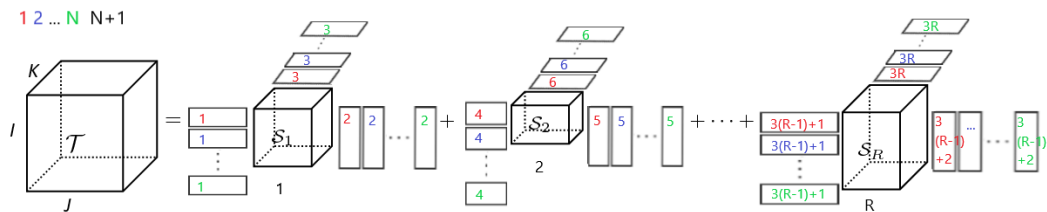


Figure 11: Order of update of method line 3

Algorithm 4 represent the pseudo-code of *line3*.

We can note that instead of solving exactly for each vector, one can compute them inexactly with a method such as conjugate gradients, like the ones included in MATLAB (*lsqr* and *cgs*). This alternative will be studied later.

4.4.3 *Als*, heuristics of the orders

The different order in which the elements are updated can be thought of as different paths taken by the algorithm in the hyperplane formed by the cost function.

If we take Figure 8, the hyperplane is formed by the function $f(x, y)$ with respect to the arguments (x, y) that are the coordinates of the 2D plane. In this example, the path of the algorithm is shown in red and minimizes first in the y direction, then in the x and keeps iterating in that order. If the algorithm would have change the order and begin by minimizing the x variable, the path would have been different, but it would not have changed the solution because the problem is convex and has only one minima which is thus global. [25]

However, in our problem, the objective function $f_{\mathcal{T}}(\dots) = \left\| \mathcal{T} - \sum_{r=1}^R \mathcal{A}_r \bullet_1 \mathbf{U}_r \bullet_2 \mathbf{V}_r \bullet_3 \mathbf{W}_r \right\|^2$ is non-convex, which means that the path taken may influence the final output of the algorithms. This is why the method *line1*, *line2* and *line3* will not produce the same output, even though they solve the same systems.

This induced the idea of finding an heuristic in the order of updates that could potentially lead to the best path.

- *steep*

The first heuristic is an alternating least squares, like the rest of the methods, but instead of having a fixed order of coordinates to minimize one after an other, the order of updates will change between each iteration. The goal is to apply the idea of a gradient descent by ordering the groups of coordinate that we will update as a function of their gradients. We are looking to solve the system of groups of coordinates from first, the one that have the steepest descent, i.e. the one with the largest norm of the gradient, to the last, the one which has the smallest one. To order the blocks, we sort them in descending order with respect of the norm of the element of gradient matrix to which they correspond. The rest of the method, i.e. the resolution of the systems, is similar to the one of the first method.

Algorithm 5 represent the pseudo-code of *steep*.

- *steep2*

The second heuristic is based on the same approach but will differ on the criteria that selects the order. Instead of using the norms of the gradients as criteria, we will base our selection on the decrease of the objective function between two steps during a gradient

method [19],

$$f(x_k) - f(x_{k+1}) \geq \frac{1}{2L} \|f'(x_k)\|_F^2$$

Where L is the Lipschitz constant.

By computing L , such that,

$$L = \frac{\|f'(x_k)\|_F^2}{2(f(x_k) - f(x_{k+1}))}.$$

In our problem, $f'(x_k)$ is the gradient of the cost function with respect to all the groups of coordinates, thus, a $(4 \times R)$ matrix, R being the number of terms in the BTD. $f(x_k) - f(x_{k+1})$ is the difference between two successive iterations and so the progression between two steps of the algorithm.

We obtain L , a $(4 \times R)$ matrix of constants that represent the optimal step of a gradient descent method.

The idea is to use the theory of the gradient method that optimize the size step of the algorithm for the decrease of the objective function and order the groups of coordinates in function of the decrease that they would create.

The order criteria will then be

$$\frac{\|\nabla_x f_{\mathcal{T}}(\dots)\|_F^2}{2L},$$

for $x = \mathbf{U}_1, \mathbf{V}_1, \mathbf{W}_1, \mathcal{A}_1, \dots, \mathbf{U}_R, \mathbf{V}_R, \mathbf{W}_R, \mathcal{A}_R$

Algorithm 6 represent the pseudo-code of *steep2*.

- *rand*

Finally, instead of having a fixed order or finding a criteria that organize it, we can try to have a random order of updates over all the groups of coordinates that changes at each iteration and comparing it to the other methods.

4.5 Complexity

The evaluation of the computational complexity of the algorithms will be done in two parts. First, the complexity of the most expensive part of the algorithm will be computed theoretically and, secondly, will be verified by empirical measures. To simplify the computation and the expression of the computational complexity, the original tensor and the core tensors are cubic (same length in three dimensions) and have respectively a side of length N and m . R denotes the number of terms in the decomposition.

4.5.1 Theoretical complexity

To compute the complexity of the *Als* method, we first look at the most influencing calls at each iterations in terms of N and m , first for the minimization of the factor matrices, then for the minimization the core tensors. The heaviest calls at each iterations are the ones that solve the systems for the updating of the factor matrices and the core tensors,

i.e. the calls that resolves Equations 10, 11, 12 and 13.

The first heavy calls are the one minimizing the factor matrices. They are done by solving a system composed by a $[N^2, m]$ matrix, the unfolded core tensor multiplied by the Kronecker product of the two other factor matrices, and a $[N^2, N]$ matrix, the unfolded main tensor, to obtain our $[N, m]$ matrices $\mathbf{U}_r$, $\mathbf{V}_r$ and $\mathbf{W}_r$ for $r = 1, \dots, R$. The second heavy calls are the minimization of the core tensors, it is done by solving a system composed by a $[N^3, m^3]$ matrix, the Kronecker product of the three factor matrices, and a $[N^3, 1]$ vector, the vectorized main tensor, to obtain a vector $[m^3, 1]$, that will then be folded to form the core tensor $\mathcal{A}_r$ of size $[m, m, m]$ for $r = 1, \dots, R$.

The complexity of these two calls is respectively $\mathcal{O}(N^3m)$ and $\mathcal{O}(N^3m^6)$. Since we work with 3 dimensional tensors, we have three factor matrices to minimize, and we have to repeat these calls R times for each terms of the decomposition. The general complexity is $\mathcal{O}(T * R(3 * (N^3m) + N^3m^6))$, then, after simplifying we obtain

$$\mathcal{O}(T * R(N^3m^6))$$

with T being the numbers of iterations.

4.5.2 Empirical measures

To measure the complexity of the algorithm, we used the profiler tool of MATLAB. The parameters N , m and R are set to a certain value and one of them is varying by multiple of itself. Then the difference between the two compilations times are compared to obtain the complexity in terms of the studied parameters.

To produce comparable results, we make sure that the number of iterations are the same between each measure of time.

First, to evaluate the complexity in terms of N , the parameters are set to $m = 5$, $R = 2$. the time measured are in seconds.

We have then:

N	time
10	15.37
20	75.36
30	281.71
40	719.64

$$\sqrt[3]{719.64/75.36} = 2.12 \approx 2$$

$$\sqrt[3]{281.71/15.37} = 2.64 \approx 3$$

It somehow matches the theoretical estimation for N .

To evaluate the complexity in terms of m , the parameters are set to $N = 20$, $R = 2$.

We have then:

m	time
5	75.36
10	2889.1

$$\sqrt[6]{2889.1/75.36} = 1.84 \approx 2$$

It somehow matches the theoretical estimation for m .

To evaluate the complexity in terms of R , the parameters are set to $N = 10$, $m = 5$.

R	time
1	7.58
2	15.78
3	24.72
4	36.56

We can see that the time of execution increase linearly as a function of R .

These results are proven by the plots in Figure 12, where the blue circles represent the running times obtained in functions of the sizes and the red curves, the polynomial functions.

$$\begin{aligned} \text{For } N \text{ we have: } t(N) &= 0.011 * N^3 + 1.056. \\ \text{For } m \text{ we have: } t(m) &= 0.0029 * m^6 + 0.9748. \\ \text{For } R \text{ we have: } t(R) &= 9.588 * R - 2.81 \end{aligned}$$

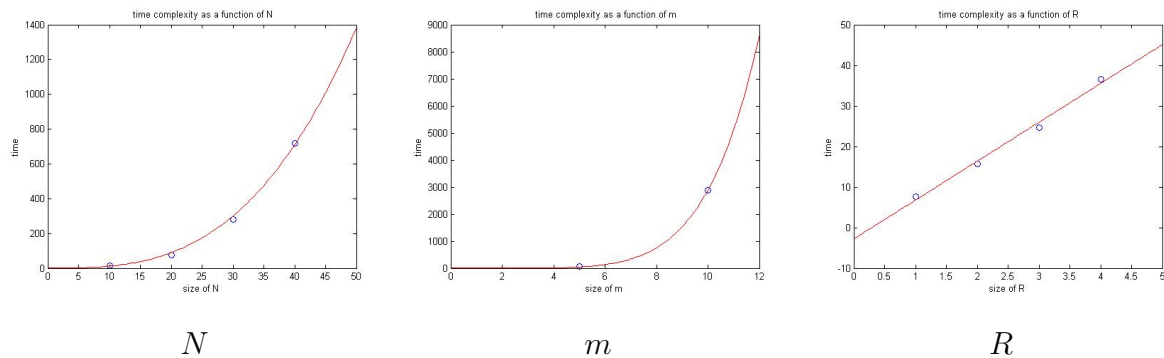


Figure 12: Time complexity of the ALS algorithm

Finally, we obtain empirically, for the full complexity,

$$\mathcal{O}(T * R(N^3 m^6)),$$

which corresponds to the theoretical estimations, T being the number of iterations.

The values used to evaluate the influence of the different parameters over the computational complexity have been limited due to a lack of memory space, larger values exceeded the allowed space so they could not have been tested.

4.6 Optimization of the algorithms

The goal of an algorithm is to find the best output possible to a certain problem, best being defined under certain criteria. But an other important aspect of an algorithm is

the time taken to obtain its outcome. Let's take two algorithms, *algo1* and *algo2*, if, for the exact same problem, they both produce the same output, or at least, the same quality of output (e.g. the same error) and the running time of *algo1* is smaller than the running time of *algo2*, then *algo1* is considered to be better than *algo2*.

It is why the algorithms implemented have been optimized to reduce, as much as possible, the computation time.

It has been done with the profiler tool of MATLAB. It helped highlighting the parts of the algorithms that take longer times and the number of times they are called.

It helped preventing redundant call such as matricizations that can be reused for several calls.

By highlighting the heaviest lines of code in terms of running time, we can try to find an other way to compute them.

For example, it helped:

- Replacing the computation of the Moore-Penrose pseudo-inverse matrices to solve systems by using the solver `\` of MATLAB.
- In the case of the computation of the core tensors, instead of using the mode- N products of Tensorlab and the pseudo-inverse, it has been replaced by the solution of the vectorized systems with the `\` solver.
- Finally, by replacing the mode- N products by the matricized products introduced previously.

The difference in computation time is shown between Figure 13 with the mode- N product and 14 with the matricized products, for the same exact problem and the same number of calls and they produced the same output, which is expected. The resolution of the matrix version is twice faster, even with the "preprocessing" of matricization of the tensors.



Figure 13: time taken with with mode- N products

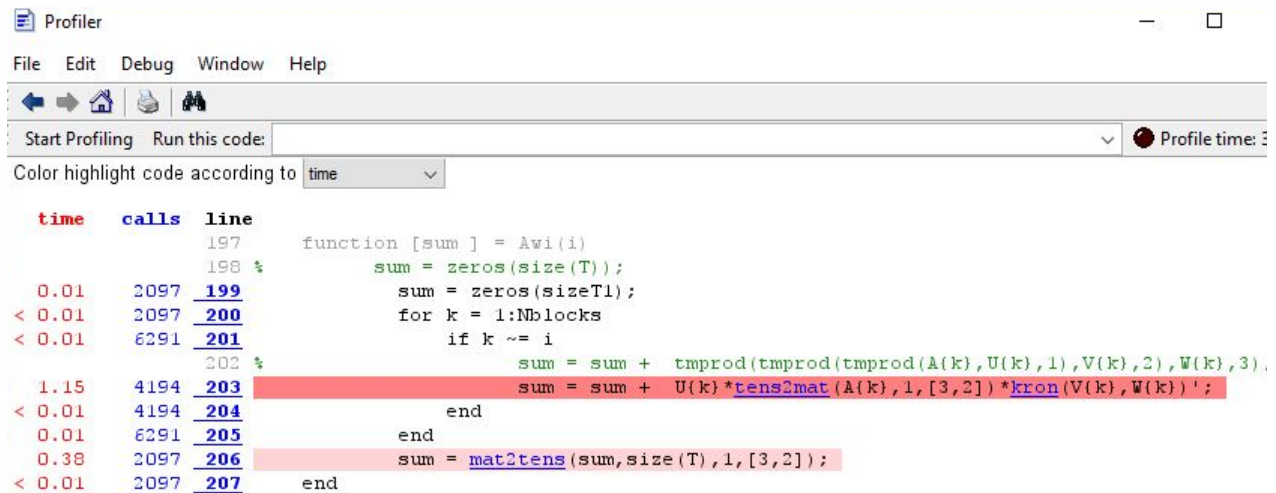


Figure 14: time taken with matricizationned products

4.7 State of the art

To the extend of our knowledge, they are not many algorithms designed to compute a BTD.

Tensorlab for MATLAB provides two functions for this purpose:

- *btd_minf* is a nonlinear unconstrained optimization algorithm that uses a quasi-Newton method with a dogleg trust region. The function will be referred as *Minf* for the rest of this thesis.
- *btd_nls* is a nonlinear least squares by Gauss-Newton algorithm with a dogleg trust region. The function will be referred as *Als* for the rest of this thesis.

Tensorlab will also be used to compute the mode- N products with the *tmprod* functions, to create the initial decompositions that will be used to test the function thanks to the *btd_rand* and *btdgen* functions.

5 Numerical results

In this section, we test the quality of the output of the algorithms previously introduced, by performing numerical tests.

We will also evaluate the influence of certain parameters and delimit certain ranges of values for these parameters. The objective is to be sure that the tests performed are relevant. As it is not possible to evaluate all the combinations of parameters, we have to delimit the problem to certain cases by excluding other ones.

To do so, we must verify that the excluded cases do not offers important insights to the problem by testing the parameters independently.

5.1 Tests

To evaluate and compare the algorithms proposed by Tensorlab and the ones introduced in this project, three elements will be taken into account, the precision by computing the relative error, the number of successes by observing the stopping criteria and the duration with the running time.

The relative error is computed by

$$err = \frac{\|\mathcal{T} - \tilde{\mathcal{T}}\|_F}{\|\mathcal{T}\|_F},$$

with $\mathcal{T}$ the main tensor that we want to decompose and $\tilde{\mathcal{T}} = \sum_{r=1}^R \tilde{\mathcal{A}}_r \bullet_1 \tilde{\mathbf{U}}_r \bullet_2 \tilde{\mathbf{V}}_r \bullet_3 \tilde{\mathbf{W}}_r$, with $\tilde{\mathcal{A}}_r, \tilde{\mathbf{U}}_r, \tilde{\mathbf{V}}_r, \tilde{\mathbf{W}}_r$, the output elements of the algorithms.

We use the relative error to evaluate the precision of the outcome instead of the absolute error ($err_{abs} = \|\mathcal{T} - \tilde{\mathcal{T}}\|_F$) to avoid scaling issues. A tensor can represent many different types of data, the scalars that compose the tensor can be a percentage of color if the tensor represent a picture or a video, they can be grading from 1 to 5 in the case of a recommendation system and so on.

In all of these cases, the norm of the tensors would not be of the same order of magnitude, and thus the absolute error between these different cases could not be compared.

The precision is a stopping criteria, as well as the main criteria for the quality of the algorithm output. For these reasons, it is measured with the relative error, to allow us to compare the results and having the same stopping conditions, regardless of the type of data that the tensor represent.

The decomposition of a tensor will be considered successful if the method does not reach the number of maximum iterations, i.e. if the norm of the gradient falls under a certain value that we will call *tolerance*, set to $1e - 12$, or if the relative error falls under a value called *precision*, set to $1e - 10$, before it reaches the maximum number of iterations set to 700.

For the tensorlab functions, a success will be considered achieved if, either, the value of the objective function falls under the set tolerance, or the norm of the steps between two iterations falls under the step size tolerance .

The time is measured by using the *tic,toc* functions of MATLAB, to obtain the running time taken by the call of the functions.

To evaluate these three criteria, a numerical test has been created. First, an original decomposition is created randomly with the function *btd_rand* of Tensorlab by choosing three parameters, the size of the main tensor, the size of the core tensors and the number of terms. Once it is created, the ground truth tensor $\mathcal{T}_{gt}$ is formed from the decomposition with the function *btdgen*. From $\mathcal{T}_{gt}$, we create the main tensor, that will be decomposed, by adding noise to form $\mathcal{T}$.

The noise is added with the function $Y = \text{noisy}(X, \text{SNR})$ which takes for arguments, X , the element(s) to add the noise to and SNR, the level of noise. Y is defined such that $Y = X + N$, with $N = \text{SNR} * \tilde{X}$, $\tilde{X}$ an element of the same size as X , generated from a standard distribution, and

$$\text{SNR} = 10 * \log_{10}((\|X\|_F^2)/(\|N\|_F^2)) \quad (14)$$

(SNR is inversely proportional to the quantity of noise added).

During a test, this procedure is applied 20 times and the algorithms are evaluated based on the three criteria introduced previously. Each time, a new original decomposition is created, thus a new ground truth tensor and a new starting point.

The starting iterates are defined by two ways, either by adding noise to the original decomposition, or by creating, randomly, a new decomposition that would not have prior information about the original decomposition, except its shape.

5.2 Tensorlab algorithms performance

Let us first test the performance of the two Tensorlab algorithms, *Nls* and *Minf*, with a noisy and a random initialization with two terms of BTD of arbitrary sizes [2, 3, 4].

The SNR between the ground-truth tensor $\mathcal{T}_{gt}$ and the main tensor $\mathcal{T}$ is set to 20 dB and between the original decomposition and the initial iterate, in the noisy case, is set to 1 dB. These will be the standard noise level, if no value are given for a test, these are the ones that are applied.

We have:

	<i>Nls</i>	<i>Minf</i>		<i>Nls</i>	<i>Minf</i>
mean error	0.0904	0.0959	mean error	0.1696	0.1861
mean time	2.5431	1.0581	mean time	25.1743	1.9015
number of successes	20/20	18/20	number of successes	11/20	7/20

Table 1: Results for Tensorlab algorithms for a BTB of 2 terms of size [2,3,4] with a noisy initial iterate (left) and a random initial iterate (right)

And with three terms of the same sizes we have:

	Nls	$Minf$		Nls	$Minf$
mean error	0.0860	0.1001	mean error	0.1584	0.1867
mean time	6.0623	1.9621	mean time	79.1931	3.0638
number of successes	20/20	14/20	number of successes	7/20	4/20

Table 2: Results for Tensorlab algorithms for a BTB of 3 terms of size $[2,3,4]$ with noisy initial iterate (left) and a random initial iterate(right)

We can observe that btd_nls , while being much slower, is more precise and has more success than btd_minf , especially in the noisy case.

We can already notice than the three terms cases are more complicated than the two terms cases, due to the longer running time and the larger mean error.

btd_nls will be used as a baseline to evaluate the method implemented in this project by comparing their performances.

5.3 Als performance

First, we want to evaluate the performance of the most simple of the methods, the alternating least square. To do so, we first perform two tests with a three terms BTB, one with a noisy initialization, an one with a random initialization to observe their convergence. We obtain:

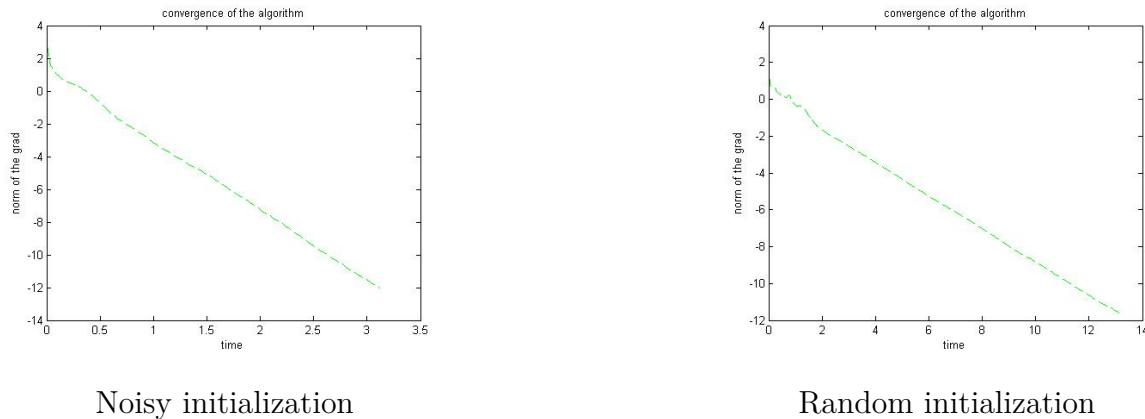


Figure 15: Norm of the gradient of Als

diff time scale The Figures 15 represent the logarithmic value of the norm of the gradients as a function of the running time of the methods.

Both cases shows that most of the convergence is done very quickly and most of the time taken by the algorithms is trying to reach the minima by advancing small decrease by small decrease.

The only notable differences between the two plots are the small irregularities in the random case at the transition between the “fast” convergence section of the firsts iterations and the “slow” convergence section that occurs for most of the running time and the important gap in running time between the two cases, the random initialization taking

much more time.

These can come from the fact that the “path” between the starting iterate and the minima, i.e. the original decomposition that we are trying to recover, is more easy for the methods to find in the noisy case than in the random case.

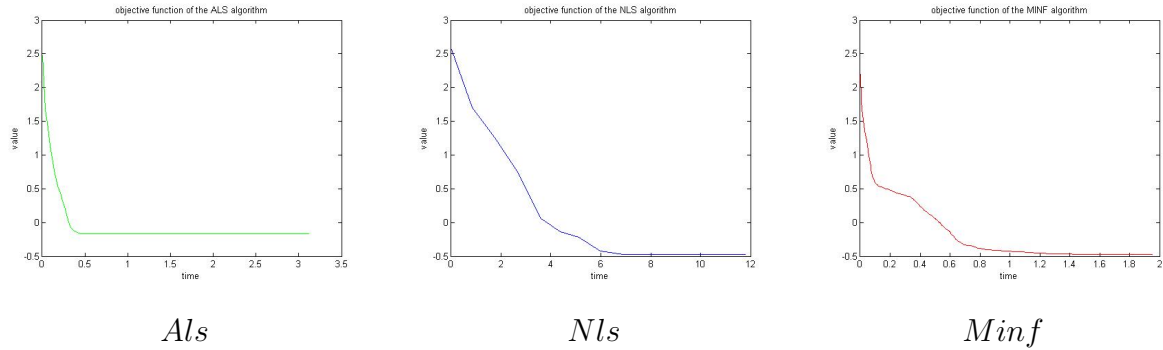


Figure 16: Convergence of the objective functions with a noisy initial iterate

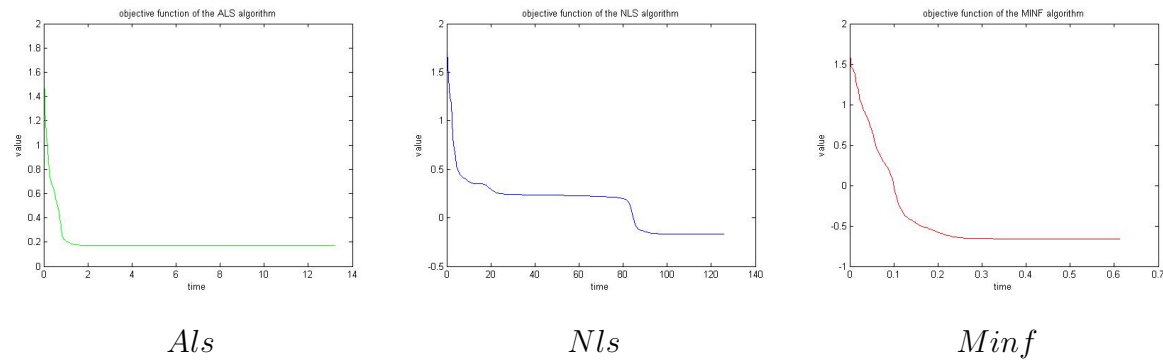


Figure 17: Convergence of the objective functions with a random initial iterate

Figures 16 and 17 represent the evolution of the logarithmic value of the cost functions of the three methods with the noisy and random starting iterate as a function of the time. These plots represent the three methods for the same example, the same main tensor to decompose and the same initial guess.

(The plots have not been placed on the same graphs for the sake of readability due to the important differences in the running time).

We can observe that the cost functions decreases rapidly during the first iterations, then slows down abruptly and almost stop converging for the rest of the iterations.

We can differentiate two sections for all the methods, a “fast” section that decreases rapidly at each iteration such that: $f(x_0) > f(x_1) > f(x_2) > \dots > f(x_n)$.

And a “slow” section, that has a much slower decrease, such that $f(x_n) \gtrsim f(x_{n+1}) \gtrsim f(x_{n+2}) \gtrsim \dots \gtrsim f(x_{end})$, for f the objective function, x_k the state of the problem at the k -th iteration and x_{end} the last iteration.

We can notice the same differences between the noisy initialization and the random initialization and the ones in the norm of the gradients plots. They are irregularities in the transition between the two sections, showing the connection between the gradient and the cost function.

This is especially true for the random case solve by the *Nls* method, when it stops converging for some iterations and then decreases again before reaching its final value.

	<i>Als</i>	<i>Nls</i>	<i>Minf</i>		<i>Als</i>	<i>Nls</i>	<i>Minf</i>
mean error	0.1104	0.0857	0.0926		0.1567	0.1584	0.1867
mean time	6.0176	5.8243	2.3172		11.3245	79.1931	3.0638
number of success	14/20	20/20	11/20		7/20	7/20	4/20
noisy initial iterate				random initial iterate			

Table 3: Results comparison between the 3 algorithms for a noisy initial iterate and a random initial iterate

5.4 Influence of different parameters

In this part, we will evaluate the influence of different parameters over the overall performance of the algorithms. That way, we will be able to test different cases and delimit the problem for certain values of the parameters so we do not have to try out all the different combination and delimit the studied cases.

5.4.1 Influence of the number of terms

First, we will assess one of the most important parameters of the BTD, the number of terms. To do so, we will compute the BTD of main tensors $\mathcal{T}$ of size $[10, 10, 10]$ in R terms of size $[2, 3, 4]$, for $R = 2, 3, 4$ with a randomly generated initial iterate. These are the default sizes for the core tensors $([2, 3, 4])$, this sizes does not ensure essential uniqueness, and the main tensors $([10, 10, 10])$, if the test does not specify them explicitly, these are the ones that will be used.

The results are displayed in Table 4.

for the sake of brevity, for the rest of the results, in tables, error, time and successes refers to the mean error, the mean time and the number of successes over the 20 runs.

	R = 2			R = 3			R = 4		
	error	time	successes	error	time	successes	error	time	successes
<i>Als</i>	0.1302	3.1866	14/20	0.1591	10.4134	5/20	0.1694	19.6383	3/20
<i>Nls</i>	0.1585	21.0444	14/20	0.1661	76.7613	5/20	0.1775	169.1185	1/20
<i>Minf</i>	0.1682	1.4758	12/20	0.1757	2.9428	3/20	0.1803	4.2179	0/20

Table 4: Performances results for 2,3 and 4 terms for a random initial iterate

Looking at the results of Table 4, we see that increasing the number of terms, as expected, increases the difficulty of the problem. This is shown, for all the algorithms, by the increase of the error and time, and the decreasing number of successes.

The increase of the number of terms increases the “degrees of freedom” of the problem, the structure of the tensor is distributed into more pieces, making it more complex and

thus, more difficult for the algorithm to recover the original decomposition that formed the main tensor.

Moreover, the cases studied in the literature rarely exceed three terms for their decompositions.

For these reasons, the tests will be limited to BTD 2-terms or 3-terms.

5.4.2 Influence of the noise on the main tensor

In this part, we study the impact of the noise added to the tensor obtained by the original decomposition to produce the main tensor, that we will try to decompose.

Let $\mathcal{T}_{gt}$ denotes the ground truth tensor, the tensor obtained by the original decomposition, and $\mathcal{T}$, the main tensor.

For this purpose, we introduce a new criterion of performance that we will call ground truth error ($error_{gt}$), defined by:

$$error_{gt} = \frac{\|\mathcal{T}_{gt} - \tilde{\mathcal{T}}\|_F}{\|\mathcal{T}_{gt}\|_F},$$

for $\tilde{\mathcal{T}}$, the tensor obtained from the output of the algorithms.

In a real case scenario, the ground truth tensor, and thus the ground truth error, are unknown. These are used in the case of created data to study the decomposition.

In reality, we do not know the real tensor and there are few chances that it equals perfectly a block term decomposition. We only know the main tensor and the noise added to the ground truth can come from the collection or the transmission of the data, or simply to represent the fact that real data does not fit perfectly a decomposition.

We test the three methods with different noise level for a 2-terms BTD with a noisy initial iterate.

For these tests, we do not evaluate the computation time because it is not much affected by the modification of this parameter and it is not relevant in its evaluation.

The results are shown in Table 5.

	SNR = 1000			SNR = 20			SNR = 5		
	error	$error_{gt}$	successes	error	$error_{gt}$	successes	error	$error_{gt}$	successes
<i>Als</i>	0.0086	0.0086	19/20	0.0948	0.0462	19/20	0.4457	0.2453	19/20
<i>Nls</i>	0.0000	0.0000	20/20	0.0908	0.4090	20/20	0.4447	0.2422	20/20
<i>Minf</i>	0.0186	0.0186	16/20	0.0912	0.0417	19/20	0.4459	0.2469	17/20

Table 5: Results obtained as a function of the noise on the ground truth tensor

Where SNR is the second argument of the *noisy* function, defined in Equation 14, such that $\mathcal{T} = noisy(\mathcal{T}_{gt}, \text{SNR})$.

We can compute the relative error, the difference, between the ground truth tensor and the main tensor with:

$$difference = \frac{\|\mathcal{T}_{gt} - \mathcal{T}\|_F}{\|\mathcal{T}_{gt}\|_F}$$

For the different levels of noise, we have:

- for SNR = 1000, $difference = 0$ (no noise)

- for $\text{SNR} = 20$, $\text{difference} = 0.1$ (mild noise)
- for $\text{SNR} = 5$, $\text{difference} = 0.5623$ (relatively important noise)

We can see from Table 5 that the noise level does not influence the number of success of the decomposition. However, it has an important impact on the error, increasing it consequently when the level of noise increase. This can be explained by the fact that the ground truth tensor is directly obtained from a perfect decomposition, whence the extremely low, if not non-existing, error obtained in the case with no added noise. When the tensor is scrambled by the noise, it loses the structure it had from the decomposition. The disturbed structure can not be found back by the algorithms, which explains the important error obtained when the ground truth tensor is noised and the direct connection between the parameters and the performance.

We can notice that the difference in mean error between the three methods comes from the non-successful runs. Runs that were successful had the same error for the three algorithm, 0 for example in the case $\text{SNR} = 1000$, the runs that reached the maximum number of iteration had a larger error, increasing the mean error and thus lowering the performance of the algorithm.

It was possible to obtain a null error for an important number of run when $\text{SNR} = 1000$ because no noise were added to the tensor obtained from the original decomposition, it means that the structure was intact and the algorithm were able to recover exactly the original decomposition.

We can notice that the ground truth error is always smaller than the error and it is a positive information because, even if we do not know the ground truth tensor in real cases, it is the one we are actually trying to decompose indirectly. This results come from the fact that the ground truth tensor is directly obtained from a BTD, thus its structure still fits perfectly a decomposition. The tensor formed by the BTD produced by the algorithm is then closer to the ground truth tensor than from the main tensor, which has lost a part of its structure due to the added noise.

We will keep the standard noise level added to the ground truth tensor to $\text{SNR} = 20$ dB now that we know the error it generates. This value is a good compromise, not too big so the noise added is too low and the case too simple, not too small so the noise added is too high and we lose the structure of the decomposition.

5.4.3 Influence of the size of the main tensor

In this part we will evaluate the impact of the size of the main tensor over the performance of the algorithms. To do so, we use three sizes of tensors, $[6, 6, 6]$, $[10, 10, 10]$ and $[15, 15, 15]$ that we decompose in 2 terms BTB with a noisy initial iterate.

We limit the case to 2 terms to avoid the problem of having more variables in the decomposition than in the main tensor that occurs with 3 terms cases and the smaller tensor sizes.

Table 6 displays the results.

We can notice in Table 6 that there are no major differences of performances between the tensor of size $[10, 10, 10]$ and the tensor of size $[15, 15, 15]$ for the same form of decomposition.

	$\mathcal{T} = [6, 6, 6]$			$\mathcal{T} = [10, 10, 10]$			$\mathcal{T} = [15, 15, 15]$		
	error	time	successes	error	time	successes	error	time	successes
<i>Als</i>	0.0989	5.1081	12/20	0.0991	1.3906	19/20	0.1023	1.4430	19/20
<i>Nls</i>	0.0762	6.7104	19/20	0.0908	2.6322	20/20	0.0956	2.4304	20/20
<i>Minf</i>	0.0951	1.6512	13/20	0.0995	1.2954	17/20	0.1012	1.6382	16/20

Table 6: Results obtained as a function of the size of the main tensor

The much worse results (in terms of running time and number of successes) obtained for the tensor of size $[6, 6, 6]$ can be due to the fact that the number of variables in the decomposition is close to the number of elements in the main tensor in comparison to the other tensors.

Elements in the small main tensor: $6^3 = 216$

Elements in the medium main tensor: $10^3 = 1000$

Elements in the large main tensor: $15^3 = 3375$

Elements in the decomposition: $2 * (2 * 3 * 4 + 2 * 6 + 3 * 6 + 4 * 6) = 156$

5.4.4 Influence of the size of the core tensors

In this section, we study the influence of the form of the decomposition, i.e. the size of the core tensors and thus the size of the factor matrices, on the performance of the algorithms.

To do so, we use a noisy initial iterate to a 2 terms block term decomposition with three different forms. The three forms are $[3, 3, 3]$, $[3, 3, 3]$, $[2, 3, 4]$, $[2, 3, 4]$ and $[5, 3, 2]$, $[2, 2, 7]$. The first one has two cubic core tensors of the same sizes and respects the conditions so the decomposition has one unique solution, the second one has two core tensor of the same size but not cubic and that does not respect the conditions of uniqueness and the third one is two core tensor of different arbitrary sizes, thus does not respects the condition for essential uniqueness.

The results are displayed in Table 7.

	Core tensors sizes								
	$[3, 3, 3], [3, 3, 3]$			$[2, 3, 4], [2, 3, 4]$			$[5, 3, 2], [2, 2, 7]$		
	error	time	successes	error	time	successes	error	time	successes
<i>Als</i>	0.0907	0.8869	20/20	0.0900	0.8619	20/20	0.0908	2.0775	19/20
<i>Nls</i>	0.0907	2.4457	20/20	0.0900	2.0642	20/20	0.0908	6.9644	20/20
<i>Minf</i>	0.1021	1.2028	16/20	0.1008	1.3910	13/20	0.0908	0.8735	19/20

Table 7: Results obtained as a function of the size of the core tensors

We can notice in Table 7 that the form of the decomposition does not seem to have a concrete influence on the performance of the algorithms.

The form has an impact on the decomposition on theory, being able to tell if a decomposition is unique or not but does not modify the numerical results with respect to running time or accuracy.

The only notable difference is the one of the running times of the last form, it can be explained by the fact that the number of elements in the third form is bigger than in the two other ones.

The size of the core tensors has no real impact on the performance under some limits, if the core tensor is a scalar or a matrix, it is not the same decomposition. They can not be too big either or the number of variables will exceed the number of elements of the main tensor.

For these reasons, the standard size for core tensor will be $[2, 3, 4]$ for the rest of the tests and is the one used if not explicitly defined.

5.4.5 Influence of the initial point

The last parameters studied will be the initial iterate. As we already noticed in the previous tests, the starting point of the algorithms will have an important impact on the final results due to the complexity of the problem and its non-convexity.

To study the impact, we will use a 3 terms BTD and, as initial iterate, a noisy version of the original decomposition, then increase the level of noise from mild to important and end with a random initial iterate, the noise level is represented by the SNR value, defined in Equation 14.

The results are displayed in Table 8.

SNR	<i>Als</i>			<i>Nls</i>			<i>Minf</i>		
	error	time	successes	error	time	successes	error	time	successes
20	0.0860	2.3176	20/20	0.0860	5.6505	20/20	0.0860	0.5148	20/20
10	0.0859	2.5417	20/20	0.0859	4.5733	20/20	0.0859	0.5291	20/20
1	0.0982	5.0902	18/20	0.0860	6.5681	20/20	0.0913	1.9998	17/20
0.1	0.1289	8.1311	11/20	0.0859	9.0396	20/20	0.0869	2.3192	15/20
0.01	0.1110	5.4570	16/20	0.0856	6.1839	20/20	0.1005	2.3989	13/20
0.001	0.1427	7.8988	12/20	0.0861	6.4632	20/20	0.0961	2.3395	12/20
0.0001	0.1459	9.6794	8/20	0.0859	6.5131	20/20	0.0947	2.2482	15/20
rand	0.1784	13.1193	4/20	0.1816	90.1333	6/20	0.1777	3.2451	5/20

Table 8: Results obtained with different initial iterate

Table 8 shows that the initial iterate has an impact on all the criteria of performances. We can now try to evaluate the obtained results by comparing the distance between the original decomposition that form the main tensor, which is the final output that we are trying to obtain, and the initial iterate, from which the algorithms start.

To do so, we must first define this distance.

Let X , the original decomposition from which the ground truth tensor is obtained, X_{init} , the starting iterate of the algorithm, $\mathcal{T}_X$ and $\mathcal{T}_{X_{init}}$, the tensors formed by the respective decompositions. The distance between the two decomposition is defined by:

$$d = \|\mathcal{T}_X - \mathcal{T}_{X_{init}}\|_F.$$

We have, for our tests, the distances:

SNR	20	10	1	0.1	0.01	0.001	0.0001	rand
d	1,7093	5,9107	25,8437	32,7950	31,6201	32,2086	33,9834	11.7847

Table 9: Mean distances between the original decomposition and the starting iterates

We can see from Tables 8 and 9 that, as expected, the further we are from the original decomposition, the worse the performance is in terms of errors, running times and number successes.

It is true until some point where adding noise does not increase the distance anymore, and we can see it in Table 8 that for these values, the results tends to stabilizes.

The exception that remains is the random initial iterate where the distance between the original decomposition and the starting point is actually smaller than the one with important noises ($\text{SNR} \leq 1$), but the results are much worse. This can be explained by the fact that the noisy cases tend to keep a part of the structure of the initial decomposition, causing the algorithms to find the best decomposition more easily. On the contrary, the random cases creates a new structure, different than the one that the algorithms is trying to find, it “blurs” more the path between the starting point and the solution, even though its distance from the original solution is shorter.

The starting iterate being a such important parameter, we will continue to test with both noisy and random initial iterates. The noisy cases will be using 1 as default noise level, it is a good compromise value, 20 does not add enough noise and the problem is too “easy”, and lower values have a similar impact to the results as shown previously. However, noisy initial iterate can be considered as an over-simplified situation. In real case scenario, the ground truth tensor and its decomposition are unknown and thus, starting from a noisy version of the solution is not feasible. For this reason, we will only test one noise level and mainly focus on the results of the random cases.

5.4.6 Conclusion on the influence of the parameters

Now that we have evaluated the impact of the different parameters of the decomposition, we have been able to delimit values for those. We will now test the performance of the different algorithms implemented in the frame of this work.

Moreover, we will use 3-terms BTM for the evaluation of the two next tests.

If we use the logic of “who can do the most, can do the least”, as we have previously

established that the 3 terms cases are more complex than the 2-terms cases, we can suppose that the results with 2-terms will be as good as, or better than with 3 terms, which is not guaranteed, and then only use this case to draw our conclusion on the performance.

5.5 *line* algorithms

First, we start testing the performance of the *line* algorithms and compare them to *Als* to verify if breaking down the problem in smaller ones improves the solutions.

The results are summarized in Table 10 and the errors plots are displayed in Figure 18.

These plots show the value of the error (y axis) as a function of the tests (x axis) for each algorithm at each test and the mean error over the 20 tests. They are useful because they display, for each run, the method or methods that have the best output, i.e. the smallest error. We can learn more information from them than from tables because we can see that even if some methods have a bigger mean error and have fewer successes, for some tests, with the same initial iterate and the same main tensor, they have a better error, thus they provide a better decomposition than algorithms that performs usually better. This information is important because it shows the complexity of the problems, even algorithms that have good performance in general can have bad results sometimes.

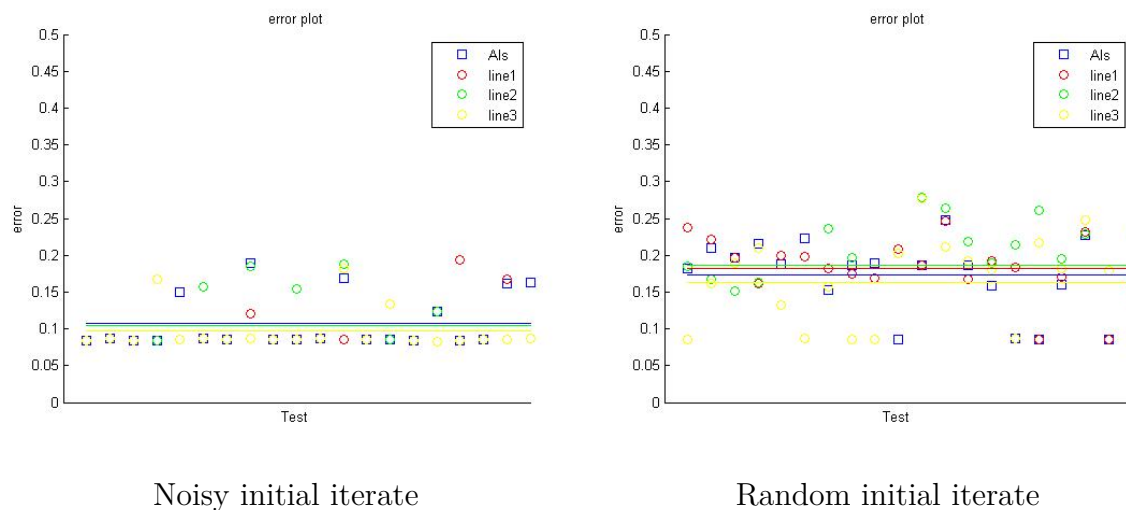


Figure 18: Error of the the *Als* and the *line* algorithms, the horizontal lines represent the mean error of the method of the corresponding color over the 20 runs.

The first observation we can make is that the mean error is bigger for the random cases for all the methods, it was to be expected and corresponds to previous results. But we can see that the best results of the random cases have the same error as the best results of the noisy cases. There are fewer of them, but it shows that, regardless of the initial state, it is possible to find what is, presumably, the best decomposition.

Now, comparing the algorithms between each other, we see that *line3* (in yellow) has, overall, the best results. This is confirmed by Table 10, which shows that *line3* has the most successes, the smaller error and the shortest computation time among the *line*

methods.

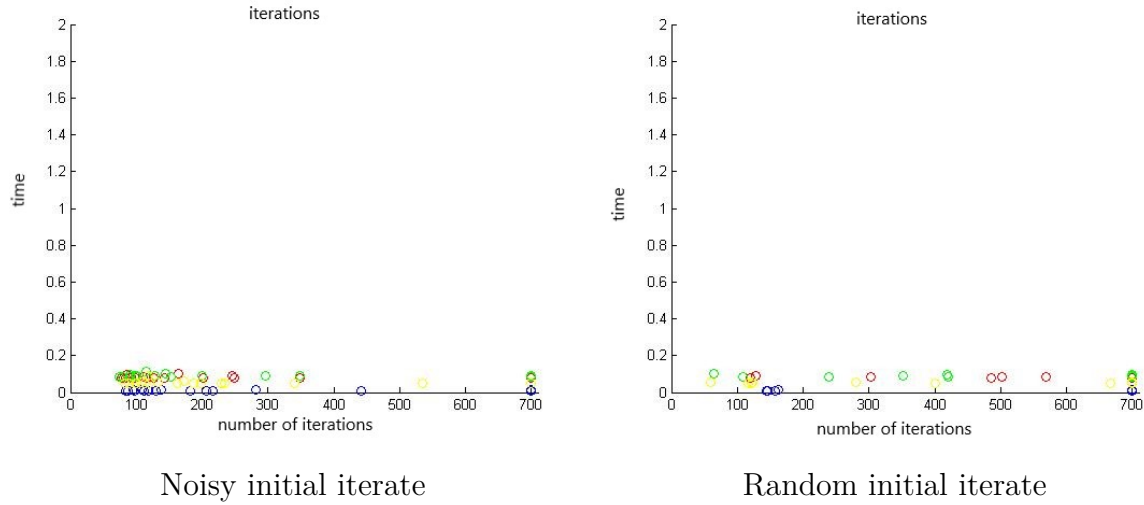


Figure 19: Iterations of the *Als* and the *line* algorithms. The color code is the same as the one in Figure 18.

We can see that a majority of circle is much smaller than 700 in the noisy case, where the majority is at 700 in the random case.

Figure 19 displays the mean time taken by the iterations as a function of the number of iterations.

We can see, by comparing the two plots, that the mean time for an iteration is the same for the noisy and the random cases, the longer running times comes from a bigger number of iterations needed in the case of a random initial iterate.

The iterations numbers that are not reaching the maximum number of iterations, 700, represent successful runs and the fact that there are more in the noisy case corresponds to the results of Table 10.

Moreover, we can see that the mean time of iterations corresponds, in general, to the computation time order shown in Table 10.

This results is counter-intuitive because MATLAB should take more time solving N systems in one time than solving one system N times. It comes from processing steps (e.g.: the matricization of the tensor, isolating the term of the BTD) that are repeated each time one line is computed where, when the matrices are computed at once, the processing steps is done once only.

	error	time	successes
<i>Als</i>	0.1076	6.7489	14/20
<i>line1</i>	0.0967	20.8807	17/20
<i>line2</i>	0.1045	27.7032	15/20
<i>line3</i>	0.0968	13.8384	18/20

error	time	successes
0.1738	12.419	4/20
0.1826	54.6888	6/20
0.1860	56.8376	6/20
0.1628	32.9523	7/20

Table 10: Results summary of the *line* algorithms for a noisy initial iterate (left) and a random initial iterate (right).

In conclusion and looking at the results Table 10, *line3* has slightly best performance in general.

Note that, as we saw, *line3* can produce, for certain runs, worse results than the other *line* methods. This is why from one test to another, the results can be different, and why we use 20 runs, because it is a value large enough to know how the algorithms will behave in most cases.

5.5.1 CG influence

We can also try to solve our systems inexactly with a conjugate gradient method instead of the solver of MATLAB. To study its performance we will create a new algorithm from *line1* and replace the " $\backslash$ " of MATLAB by conjugate gradient and then compare the performance of the two approaches. The choice of *line1* is arbitrary, we could have taken any *line* method and apply the same conclusions.

We can use the methods provided by MATLAB (e.g.: *cgs* or *lsqr*) to solve a system such that $\mathbf{x} * \mathbf{A} = \mathbf{b}$.

Where, in our case, $\mathbf{x}$ are the lines of the factor matrices, $\mathbf{A}$ is the unfolded core tensor of the same term multiplied by the two other factor matrices and $\mathbf{b}$ the corresponding slice of unfold main tensor subtracted by the other terms of the BTD.

To optimize the method and decrease the running time, the initial iterate used is the solution of the previous iteration.

Because $\mathbf{A}$ is not squared, to use *cgs* we have to modify the system into $\mathbf{x} * \mathbf{A} * \mathbf{A}^T = \mathbf{b} * \mathbf{A}^T$. The algorithm that solves the systems with conjugate gradient methods is called *line1_CG*. The parameter we can modify to have an impact on the results of the methods is the tolerance of the conjugate gradient, i.e. its final error. Starting by applying a low tolerance ($> 1e - 5$), we obtain the same results as the one solved exactly but with a longer running time.

Then, applying a higher value as tolerance, the solutions of the systems are not precise enough and the errors obtained are strictly bigger than the ones obtained with " $\backslash$ ".

The results produced by these two approaches do not improve the method and are strictly worst, thus it is not interesting to study them further.

Table 11 displayed the results obtained with two tolerances for the conjugate gradient, $1e-2$ for a large one, and $1e-7$ for a small one.

The case used is a 2-terms BTD with a random initialization.

	<i>line1</i>	<i>line1_CG</i>		<i>line1</i>	<i>line1_CG</i>
mean error	0.1556	0.1570	mean error	0.1624	0.1624
mean time	0.0481	0.0745	mean time	0.0441	0.0968

Table 11: Compared results for the *line1* and *line1_CG* method with a large tolerance (left) and a small tolerance (right) for the conjugate gradient method

Finally, a last tolerance we can use is a variable one, by decreasing it at each iteration, we can start with a big value (0.5) and end with a small one ($1e - 12$). That way, we begin by approaching the solution rapidly without spending too much time trying to get very close in the first steps, but approaching them very precisely in the last iterations. The results are shown in Figure 20.

We see that this approach is equal or worst, in all runs, for the noisy case, but manages to have better results than the original *line1* method for some runs in the random case, which is an interesting result.

But the overall performances are not as good so we will keep solving the system exactly.

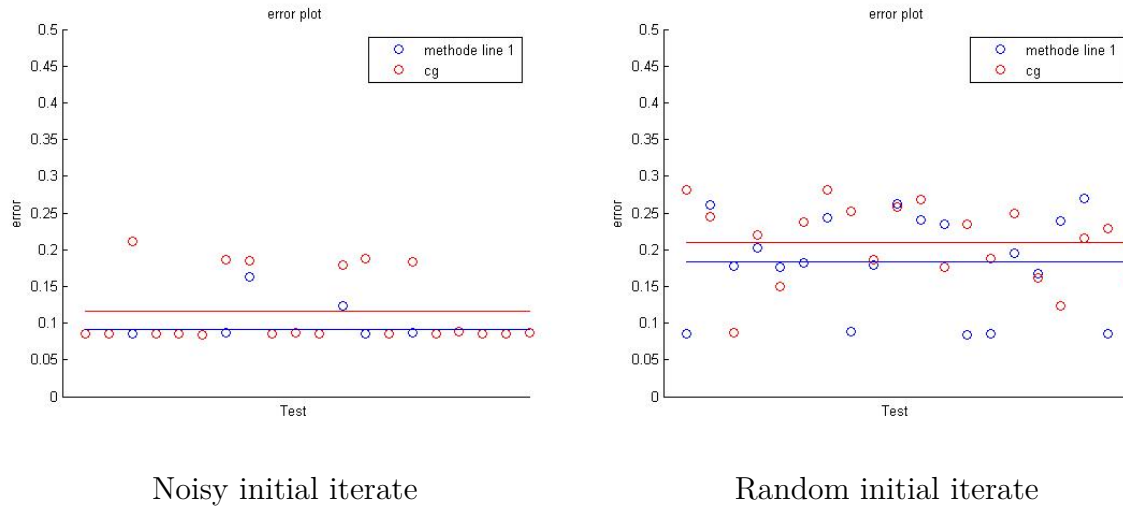


Figure 20: Error plots of the *line1* algorithms using $\backslash$ and a conjugate gradient

5.6 *steep* and *rand* algorithms

Second, we can test the performances of the algorithms with different heuristics for the order, i.e. *steep*, *steep2* and *rand*, and compare them with *Als*.

The goal of these algorithms is to organize the order of alternating least squares methods to find the right path, or at least a better path than the fixed order of the *Als* algorithm.

The results are summarized in Table 12 and the errors plots are displayed in in Figure 21.

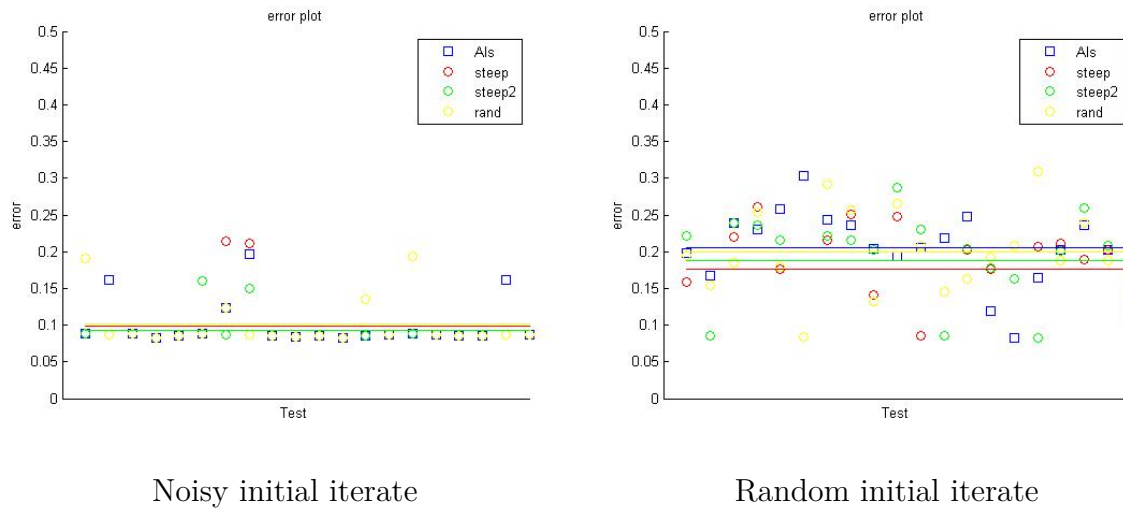


Figure 21: Error of the different order heuristic algorithms, the horizontal lines represent the mean error of the method of the corresponding color over the 20 runs.

By observing the plots, we can draw the same conclusions as the ones of the *line* methods as for the differences between the noisy cases and the random cases.

We can see that the results of *rand* are very close to the ones of *Als*. This means that a fixed order chosen arbitrarily is not more efficient than any method with no specific

order, i.e. where the order is chosen randomly.

We can notice that the *steep* and *steep2* algorithms provide better results, especially for the random cases. This can be explained by the fact that the *steep* methods goes first in the direction of the steepest coordinate, possibly orienting the path towards the best solution, what the other two methods are not doing. The noisy initial iterate has, while being further from the solution, a more direct path than the random initial iterate, there is a lesser need to be oriented in the right direction then. This could explain the differences between the two cases for the methods.

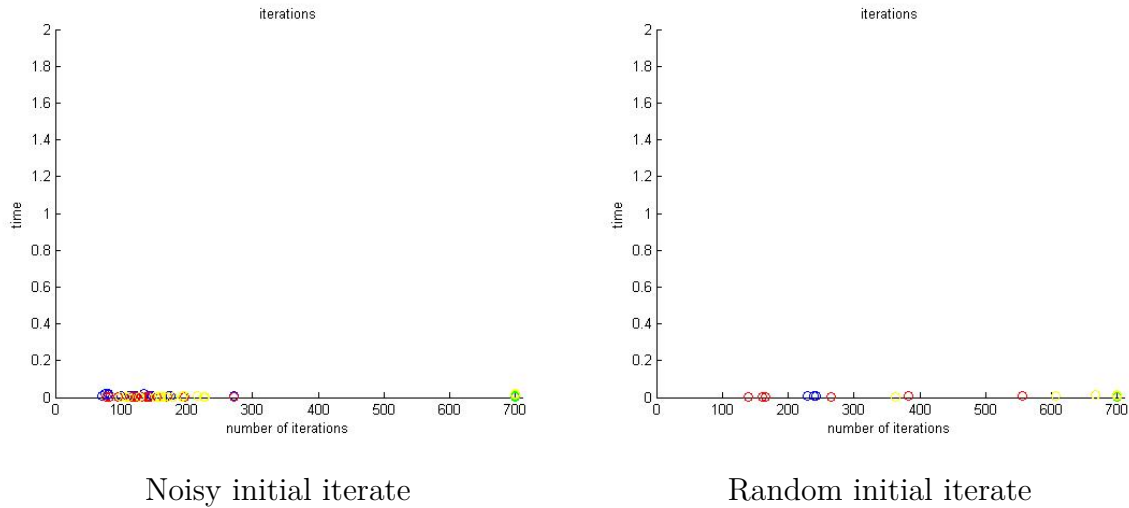


Figure 22: Iterations of the different order heuristics algorithms

We can observe in Table 12 that the total running time of the different methods are close. It is shown in Figure 22 that the time taken by each iteration is similar for the four algorithms and that they have approximately the same number of iterations. This means that the processing steps that orders the methods have a low impact on the full computation time.

	error	time	successes
<i>Als</i>	0.1009	5.4195	16/20
<i>steep</i>	0.0987	4.5707	18/20
<i>steep2</i>	0.0929	5.6256	18/20
<i>rand</i>	0.1009	6.7877	16/20

	error	time	successes
	0.2054	11.8495	3/20
	0.1768	11.8950	6/20
	0.1874	11.5668	5/20
	0.1993	13.7769	3/20

Table 12: Results summary of the different heuristics

In conclusion, the general performance is somewhat similar to the one of the previous in terms of error and number of successes.

We can nonetheless notice that the *steep* and *steep2* methods have better results in general, looking at Table 12 and Figure 21.

But again, the results are almost impossible to predict and the performance can change from one run to an other and the goal is to watch the overall successes of the algorithms.

5.6.1 Influence of the orthogonality of the matrices

During the computation of the decomposition we use the QR decomposition on the factor matrices to obtain orthogonal matrices. The goal is to obtain a final decomposition with orthogonal factor matrices. To study the impact of the factorization, we will use the Als method with and without the decomposition to four different cases.

The two compared methods are Als and Wqr , the last one is the exact same algorithm as Als but the factor matrices are not factorized into orthogonal matrices.

The cases are differentiated by the original decomposition, whether the factor matrices are orthogonal or not, and by their initial iterates, if it is also orthogonal or not, they will be called, respectively, orthogonal origin and orthogonal initialization cases.

The results are displayed in Tables 13 and 14.

	error	time	successes
Als	0.1507	9.7907	9/20
Wqr	0.1761	11.2114	6/20

initial iterate with orthogonal
factor matrices

error	time	successes
0.1583	8.9128	10/20
0.3793	13.3072	0/20

initial iterate without
orthogonal factor matrices

Table 13: Results for an original decomposition with orthogonal factor matrices

	error	time	successes
Als	0.0780	12.7198	0/20
Wqr	0.0802	13.2086	0/20

initial iterate with orthogonal
factor matrices

error	time	successes
0.0790	13.0087	0/20
0.0792	13.4493	0/20

initial iterate without
orthogonal factor matrices

Table 14: Results for an original decomposition without orthogonal factor matrices

In this case, we will mainly focus on the relative error and the number of successes to study the influence of the QR factorization.

First, one can notice that in most of the cases, the relative error and the number of successes are quite close when the original decompositions do not have orthogonal factor matrices, but we do not have any successful runs.

The case with the total largest number of success is the one with an orthogonal origin and a non-orthogonal initialization case. It can be explained fact that we want to decompose a tensor obtained by orthogonal factor matrices, starting from an iterate without orthogonal factor matrices. The algorithm that factorize its factor matrices easily obtain matrices that are similar to the ones that created the ground truth tensor due to the fact that it returns orthogonal matrices, what the other algorithms does not. This can explain the important difference in terms of quality of the outcomes.

We can notice that none of the methods has successes with the non-orthogonal origin cases. This motivates the use of orthogonal factor matrices in the original decomposition for the other tests of performances.

We see that, in general, there is no improvement in the results when using a method without factorization, this is why we will keep using factorization in the algorithm for the rest of the methods.

5.7 Final results

Now that we have fixed the parameters and tested the different methods, we can compare the method that produced the best results with *Als* and *Nls* to evaluate their performances regarding the state of the art.

The results are displayed in Figures 23, 24 and in Tables 15, 16.

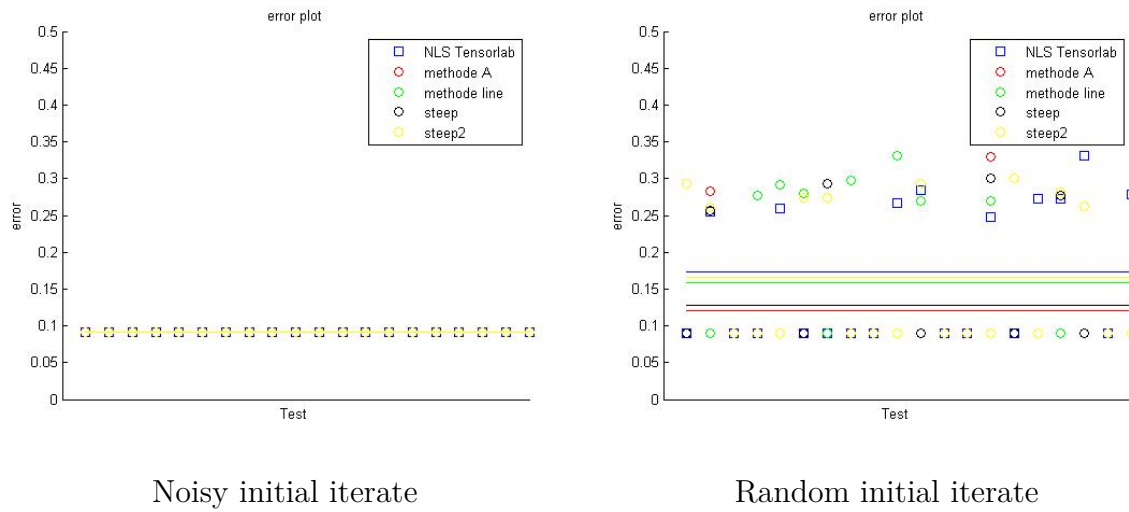


Figure 23: Error plots for 2-terms BTM, the horizontal lines represent the mean error of the method of the corresponding color over the 20 runs.

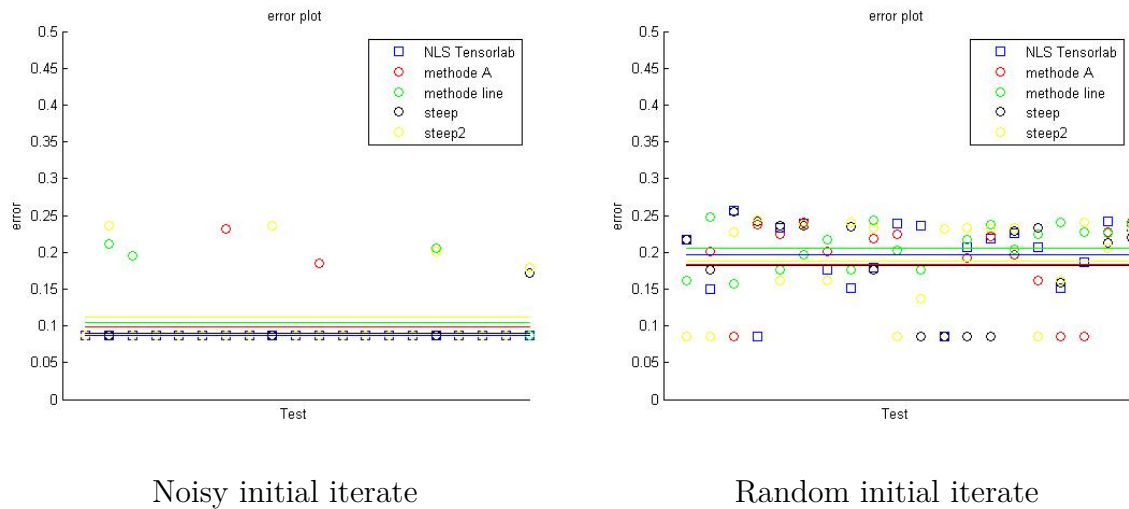


Figure 24: Error plots for 3-terms BTM, the horizontal lines represent the mean error of the method of the corresponding color over the 20 runs.

First, we can notice, by comparing the results between the noisy and random case and between 2-terms and 3-terms BTM, that they correspond to the ones previously obtained.

	error	time	successes
<i>Nls</i>	0.0907	2.1909	20/20
<i>Als</i>	0.0907	0.8196	20/20
<i>line3</i>	0.0907	2.4959	20/20
<i>steep</i>	0.0907	0.9378	20/20
<i>steep2</i>	0.0907	0.9551	20/20

Noisy case

error	time	successes
0.1727	25.2991	11/20
0.1206	2.4320	17/20
0.1590	10.4376	13/20
0.1280	3.8546	15/20
0.1657	5.4277	12/20

Random case

Table 15: Results for a 2-terms decomposition.

	error	time	successes
<i>Nls</i>	0.0864	5.3812	20/20
<i>Als</i>	0.0985	2.9181	18/20
<i>line3</i>	0.1040	8.8751	18/20
<i>steep</i>	0.0906	2.8536	19/20
<i>steep2</i>	0.1117	4.4825	16/20

Noisy case

error	time	successes
0.1960	93.6574	3/20
0.1840	10.3895	6/20
0.2049	37.8090	2/20
0.1827	11.7198	5/20
0.1877	12.9303	4/20

Random case

Table 16: Results for a 3-terms decomposition.

The noisy and the 2-terms cases have, for each criteria, better results than the random and the 3-terms cases.

On the one hand, 2-terms BTM with a noisy initial iterate is arguably the simplest case, it is successful for all the methods for most of the time and here had 100 percent of success. We can see that no algorithms has been “stuck” in a local minima or a saddle point because they all had, essentially, the same outputs.

On the other hand, the 3-terms BTM with a random initial iterate is the most difficult case that, in average for all the algorithms, 20 percent of success and have an average error more than twice as big as the average error of the 2-terms noisy case.

Now we can compare the results obtained by the different methods. First with the *Nls* algorithm, it has a 100 percent success for the two noisy cases and the lowest average error. As long as we will have access to a noisy version of the decomposition that we want to recover, even with an important level of noise, as seen in section 5.4.5, *Nls* will be the best method.

But a noisy starting iterate is almost impossible to obtain in most cases and the results produced by *Nls* for the random cases are not as good. It has one of the highest average error, one of the lowest number of successes and is the slowest method. For these reasons, when using a randomly generate initial iterate, *Nls* will not often produce the best solution.

Second, the *line3* method can produce the best results, as seen in the second run of the random case in Table 23, but has overall a large number of unsuccessful runs and is the slowest of the alternating least squares methods. For these reasons, the *line3* algorithm is not the best method.

Third, *steep2* is producing good results in quite short times but has overall less successes and a higher average error so this method will not be preferred to the last two.

Finally, the results produced by *Als* and *steep* are quite close in terms of numbers of successes and average error and running time. They are the ones producing the best decompositions for the random cases but, looking at the results of section 6.6, the *steep* method will be preferred as the most efficient algorithm.

5.8 Multistarts

After comparing the different methods, we attested the impact of the starting iterate, but, as previously mentioned, using a noisy version of the original decomposition as initial point is not possible in real cases.

However, we noticed that for some runs, it was possible to succeed with a random initial iterate. The idea is then to find the best block-term decomposition of a tensor by trying different random starting points and selecting the solution that had the lowest relative error. To do so, we will try this approach with the *steep* method due to its relatively good results.

To test this technique, we will produce an original decomposition and from it, form the ground truth tensor to what we will add noise to obtain the the main tensor, like the previous numerical tests. Now, we will decompose the main tensor with the *Nls* algorithm, using a noisy original decomposition as initial guess. The error of the obtained solution will be used as reference because, throughout all the test, *Nls* and a noisy initial iterate produced the best decomposition possible at every run. After that, we use different random starting points with *steep* and keep only the best solution, i.e. the one with the lowest error, and compare it to the reference.

We saw in Table 16 that for a 3-terms BTB, *steep* succeed in, on average, 20 to 25 percent of the runs, for that reason, we could use 5 as the number of different starting positions, but we will be using 7 to increase the number of success.

The results are displayed in Tables 17 and 18, where reference is the relative error obtained with *Nls* and a noisy initialization, best error is the error minimum obtained over the 7 runs and the running time is the time taken by the 7 runs.

reference	0,0888	0,0907	0,0915	0,0896	0,0909	0,0919	0,0911	0,0895	0,0908	0,0897
best error	0,0888	0,0907	0,0915	0,0896	0,0909	0,0919	0,0911	0,0895	0,0908	0,0897
running time	31,21	46,34	52,33	44,58	21,35	36,12	55,79	38,26	37,10	36,01

Table 17: results with multiple trials for the *steep* algorithm on a 2-terms BTB

reference	0.0854	0.0847	0.0842	0.0860	0.0879	0.0855	0.0864	0.0846	0.0871	0.0854
best error	0.0854	0.1615	0.0842	0.0860	0.0879	0.0855	0.0864	0.0846	0.0871	0.0854
running time	103.33	111.95	103.92	87.66	84.39	115.90	89.11	105.10	103.02	103.85

Table 18: results with multiple trials for the *steep* algorithm on a 3-terms BTB

The error between the references values and the best error that are the same in the Table is lower than $1e - 12$.

We can notice that there is only one test, the second one, for which this technique has failed for the 3-terms decomposition, while succeeding every time for a 2-terms decomposition.

We only focus here on the best solutions, i.e. the ones that produce the lowest error, because if we have the choice between several decompositions, we will naturally choose the best one and we will not pay too much attention to the unsuccessful runs.

This approach may not be efficient and is not supported by any theory but is able to produce effective results. We see in Table 17 and 18 that in the vast majority of the cases, the error is the same between the noisy and the random case. These results are quite positive because they have been obtained without any knowledge of the original decomposition.

Moreover, we can see by comparing Tables 17, 18 and Tables 15, 15 that the running times are not much slower than *Nls* in random cases, even by completing 7 runs. The reference times for the *Nls* method are the average times taken by the method for one run in the random cases, they are displayed in Tables ?? and ?? and are equal to 25.1743 for a 2-terms BTB and 79.1731 for a 3-terms BTB.

Note that the number of starting positions can be thought of as a parameter that can be increased when the case is more complex to have more chances of success but the method will be slower, or decreased in simpler cases to accelerate it but with a higher risk that the method will not succeed.

5.9 Interpretation of the results

Now that all the methods have been tested for all the wanted values of the parameters, we can draw some conclusions and interpret the results.

First, we saw in section 5.4.1 that increasing the number of terms increased consequently the difficulty of the problem. By adding new variables and new components to the decomposition, the structure that formed the main tensor becomes so complicated that, with “only” 4 terms, the methods have almost no successes.

Moreover, as shown in section 5.4.2, it is impossible to recover exactly a decomposition from a tensor when noise has been added to the ground truth tensor. The noise blurs the structure of the decomposition, thus creating the relative error between the tensor formed by the output BTB and the main tensor. We also noticed that the ground truth error was smaller than the relative error. This means that, by decomposing the main tensor, we denoised it partially, i.e. decreased the noise level, to recover, to a certain extent, the ground truth tensor.

Finally, in section 5.4.5, we noticed the importance of the starting point. The results obtained with a random initial iterate are worst in every criteria than the ones obtained with a noisy initial iterate.

To make a short analogy, the solution of the problem, the original decomposition that we want to recover, would be the destination. The noisy case puts the starting iterate on a rather direct path where the chances of getting lost are low, even when we are further from the destination, as seen in Table 9. The random case puts the starting iterate on a winding path with bifurcations where there is a higher chance of getting lost, even when

the starting point is closer from the destination.

But we also noticed that the best results, the ones that have the lowest error, from the two initial cases have approximately the same error. This means that, as expected, the initial guess, from which the algorithms starts, does not influence the best output, it is just more difficult to obtain.

As we saw throughout the results of this thesis, the results depends highly on the run, especially for the random cases, mainly due to the different starting iterate. This gives the impressions that these results are random, because, for different reasons, the methods do not reach the decomposition that fits the most in the least square sense, i.e. the global minima of the cost function.

The reasons for which the algorithms do not attain the global minima can be multiple, first, even if the run the norm of the gradient falls under the set tolerance, it does not mean that that the method has reach the best decomposition, the problem being non-convex, it can be “stuck” in a local minima or a saddle point.

But convergence is not always witnessed, for some runs, the algorithms reaches first the maximum number of iteration before the norm of the gradient falls under the tolerance. This can be explained by a phenomenon called “swamps” as studied in [10], [17] and [18]. These effects occurs when the cost function almost stops decreasing for a certain, sometimes large, number of iteration, before quickly decreasing in a few iterations steps. This can forbid the algorithm to find a minima under the set number of maximum iterations.

6 Conclusion

Throughout this work, we studied the decomposition of higher-order tensor, and more specifically, the block term decomposition of third-order tensors.

Tensors are gaining popularity in multiple fields of applied mathematics for their ability to synthesize several data set of lower-order tensors like matrices and vectors into one mathematical object, i.e. the tensorization of the data. Their structure being more complex, shown by the mode- n products and mode- n ranks, allow us to create a larger variety of decompositions that could provide more information about the data.

The BTM is sharing specificities of two other popular decomposition, the CP and the Tucker decomposition, thus making it more versatile and usable in more situations.

In this work, we focused on implementing an algorithm to compute the best BTM of a tensor in the least square sense and proceeded by coordinate descent. From this perspective, we compared state of the art algorithms from Tensorlab to a regular alternating least square method and then tried to improve it by different ways.

A first approach was to break the groups of coordinates that were the factor matrices into smaller groups of coordinates that were vectors. This way, we had the possibility to alternate in different orders and producing different outputs, thus highlighting the importance of the order in the solution of the problem and inducing the idea of finding an order that would be more efficient.

In a second time, we implemented methods that changed the order of alternation at each iteration with a selecting criteria that depends on the gradients of the cost function.

The problem of finding a BTM is complex due to several aspects. The larger number of components that compose the BTM make it more difficult to compute than the Tucker decomposition. Moreover, the non-convexity of the cost function causes the algorithms to get “stuck” on local minima or saddle points preventing it to reach the optimal solution. Finally, we highlighted in this work the impact of the initial iteration on the results of the algorithms. Using a noisy version of the original decomposition is not often possible in real cases, it is why developing a method that could provide a good initial guess would be a major improvement in this domain.

With this in mind, we opted for several starting points to decompose a tensor so that we increase the chances to reach the optimal solution.

While we do not have tried all the different possible orders, it is doubtful that one single selection criteria can solve the problem and find the best decomposition at each runs. However, there are still some room for improvement that could be explored.

For example, on the continuation of this work, one could use methods to reduce the swamps phenomenon that have been recently implemented to enable the algorithms to reach a minima more often. Finally, putting the coordinate descent strategy aside, one could also solve the decomposition with parallel computing, by distributing the different groups of coordinate and solving them in parallel, accelerating the algorithms and, possibly, improving the results.

7 Bibliography

References

- [1] Anima Anandkumar. *Role of Tensors in Machine Learning*. Technologie. URL: <https://fr.slideshare.net/AnimaAnandkumar/role-of-tensors-in-machine-learning> (visited on 07/23/2019).
- [2] Brett W. Bader and Tamara G. Kolda. “Efficient MATLAB Computations with Sparse and Factored Tensors”. en. In: *SIAM Journal on Scientific Computing* 30.1 (Jan. 2008), pp. 205–231. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/060676489. URL: <http://epubs.siam.org/doi/10.1137/060676489> (visited on 08/17/2019).
- [3] Andrzej Cichocki et al. “Tensor Decompositions for Signal Processing Applications From Two-way to Multiway Component Analysis”. In: *Signal Processing Magazine, IEEE* 32 (Mar. 2014). DOI: 10.1109/MSP.2013.2297439.
- [4] *Coordinate descent*. en. Page Version ID: 902603730. June 2019. URL: https://en.wikipedia.org/w/index.php?title=Coordinate_descent&oldid=902603730 (visited on 07/18/2019).
- [5] L. De Lathauwer. “Decompositions of a Higher-Order Tensor in Block Terms—Part II: Definitions and Uniqueness”. In: *SIAM Journal on Matrix Analysis and Applications* 30.3 (Jan. 2008), pp. 1033–1066. ISSN: 0895-4798. DOI: 10.1137/070690729. URL: <https://epubs.siam.org/doi/10.1137/070690729> (visited on 07/18/2019).
- [6] Lieven De Lathauwer. “Decompositions of a Higher-Order Tensor in Block Terms—Part I: Lemmas for Partitioned Matrices”. In: *SIAM J. Matrix Anal. Appl.* 30.3 (Sept. 2008), pp. 1022–1032. ISSN: 0895-4798. DOI: 10.1137/060661685. URL: <http://dx.doi.org/10.1137/060661685> (visited on 07/18/2019).
- [7] Lieven De Lathauwer and Dimitri Nion. “Decompositions of a Higher-Order Tensor in Block Terms—Part III: Alternating Least Squares Algorithms”. In: *SIAM J. Matrix Analysis Applications* 30 (Jan. 2008), pp. 1067–1083. DOI: 10.1137/070690730.
- [8] Zhen-Hua Feng et al. “A Unified Tensor-based Active Appearance Face Model”. en. In: (Dec. 2016). URL: <https://arxiv.org/abs/1612.09548v2> (visited on 07/23/2019).
- [9] Xiurui Geng et al. “A High-Order Statistical Tensor Based Algorithm for Anomaly Detection in Hyperspectral Imagery”. en. In: *Scientific Reports* 4 (Nov. 2014), p. 6869. ISSN: 2045-2322. DOI: 10.1038/srep06869. URL: <https://www.nature.com/articles/srep06869> (visited on 07/23/2019).
- [10] Xue Gong, Martin J. Mohlenkamp, and Todd R. Young. “The Optimization Landscape for Fitting a Rank-2 Tensor with a Rank-1 Tensor”. In: *SIAM J. Applied Dynamical Systems* 17 (2018), pp. 1432–1477. DOI: 10.1137/17M112213X.
- [11] Niao He and Lucas Buccafusca. “Lecture 12: Coordinate Descent Algorithms”. en. In: (), p. 7.
- [12] Borbála Hunyadi et al. “Block term decomposition for modelling epileptic seizures”. en. In: *EURASIP Journal on Advances in Signal Processing* 2014.1 (Sept. 2014), p. 139. ISSN: 1687-6180. DOI: 10.1186/1687-6180-2014-139. URL: <https://doi.org/10.1186/1687-6180-2014-139> (visited on 07/18/2019).

-
- [13] Mariya Ishteva et al. “Tucker compression and local optima”. In: *Chemometrics and Intelligent Laboratory Systems*. Multiway and Multiset Data Analysis 106.1 (Mar. 2011), pp. 57–64. ISSN: 0169-7439. DOI: 10.1016/j.chemolab.2010.06.006. URL: <http://www.sciencedirect.com/science/article/pii/S0169743910001164> (visited on 07/18/2019).
 - [14] Tamara G. Kolda and Brett W. Bader. “Tensor Decompositions and Applications”. en. In: *SIAM Review* 51.3 (Aug. 2009), pp. 455–500. ISSN: 0036-1445, 1095-7200. DOI: 10.1137/07070111X. URL: <http://epubs.siam.org/doi/10.1137/07070111X> (visited on 07/18/2019).
 - [15] Yong Li, Wenrui Dai, and Hongkai Xiong. “Compressive Tensor Sampling with Structured Sparsity”. In: *2016 Data Compression Conference (DCC)* (2016), pp. 181–190. DOI: 10.1109/DCC.2016.16.
 - [16] Ajmal Mian, Mohammed Bennamoun, and R. A. Owens. “Matching Tensors for Pose Invariant Automatic 3D Face Recognition”. In: *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops* 0 (June 2005), p. 120. ISSN: 0-7695-2660-8. DOI: 10.1109/CVPR.2005.490.
 - [17] M. Mohlenkamp. “The Dynamics of Swamps in the Canonical Tensor Approximation Problem”. In: *SIAM Journal on Applied Dynamical Systems* (Jan. 2019), pp. 1293–1333. DOI: 10.1137/18M1181389. URL: <https://epubs.siam.org/doi/abs/10.1137/18M1181389> (visited on 08/10/2019).
 - [18] Carmeliza Navasca, Lieven De Lathauwer, and Stefan Kindermann. “Swamp reducing technique for tensor decomposition”. In: *European Signal Processing Conference* (Jan. 2008).
 - [19] Yu Nesterov. *Introductory Lectures On Convex Programming*. 1998.
 - [20] Guillaume Olikier. “Tensor approximation by block term decomposition”. Master Thesis. 2017.
 - [21] Guillaume Olikier, P.-A. Absil, and Lieven De Lathauwer. “Variable Projection Applied to Block Term Decomposition of Higher-Order Tensors”. en. In: *Latent Variable Analysis and Signal Separation*. Ed. by Yannick Deville et al. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 139–148. ISBN: 978-3-319-93764-9.
 - [22] Kaare Brandt Petersen and Michael Syskind Pedersen. “[<http://matrixcookbook.com>]”. en. In: (), p. 72.
 - [23] B. Sheehan and Y. Saad. “Higher Order Orthogonal Iteration of Tensors (HOOI) and its Relation to PCA and GLRAM”. In: *Proceedings of the 2007 SIAM International Conference on Data Mining*. Proceedings. Society for Industrial and Applied Mathematics, Apr. 2007, pp. 355–365. ISBN: 978-0-89871-630-6. DOI: 10.1137/1.9781611972771.32. URL: <https://epubs.siam.org/doi/abs/10.1137/1.9781611972771.32> (visited on 07/18/2019).
 - [24] Andrews Sobral. “Robust Low-rank and Sparse Decomposition for Moving Object Detection: From Matrices to Tensors”. PhD thesis. May 2017. DOI: 10.13140/RG.2.2.33578.82884.
-

-
- [25] J. M. Soriano. “Global minimum point of a convex function”. In: *Applied Mathematics and Computation* 55.2 (May 1993), pp. 213–218. ISSN: 0096-3003. DOI: 10.1016/0096-3003(93)90022-7. URL: <http://www.sciencedirect.com/science/article/pii/0096300393900227> (visited on 08/16/2019).
- [26] *Tensorlab / A Matlab package for tensor computations*. URL: <https://www.tensorlab.net/> (visited on 08/17/2019).
- [27] Jinmian Ye et al. “Learning Compact Recurrent Neural Networks with Block-Term Tensor Decomposition”. In: *arXiv:1712.05134 [cs, stat]* (Dec. 2017). arXiv: 1712.05134. URL: <http://arxiv.org/abs/1712.05134> (visited on 07/18/2019).

8 Annexes

8.1 Matlab implementations

During this thesis, different MATLAB functions and scripts have been implemented and used, you can find them here:

8.1.1 functions implemented in this thesis to solve a BTM

- **btd_Als.m**: the function corresponding to the *Als* algorithm.
- **btd_line1.m**: the function corresponding to the *line1* algorithm.
- **btd_line2.m**: the function corresponding to the *line2* algorithm.
- **btd_line3.m**: the function corresponding to the *line3* algorithm.
- **btd_steep.m**: the function corresponding to the *steep* algorithm.
- **btd_steep2.m**: the function corresponding to the *steep2* algorithm.
- **btd_rand.m**: the function corresponding to the *rand* algorithm.
- **btd_sqf.m**: the function compared to *Als* to evaluate the impact of the factorization.
- **btd_line1_CG.m**: the function compared to *line1* to evaluate the impact of the conjugate gradient.

8.1.2 scripts implemented in this thesis to test the performance of the algorithms

- **testTensorlab.m**: script that test the methods *Nls*, *Minf* and *Als*.
- **testlines.m**: script that test the methods *line1*, *line2*, *line3* and *Als*.
- **teststeep.m**: script that test the methods *steep*, *steep2*, *rand* and *Als*.
- **testfinal.m**: script that test the methods *Nls*, *Als*, *line3*, *steep* and *steep2*.

- **Trials.m**: script that evaluate the results when using several starting point.
- **influenceQF.m**: script that evaluates the influence of the factorization of the factor matrices.
- **influenceCG.m**: script that evaluates the influence of the conjugate gradient to solve the systems.
- **plotcomplexity.m**: script that produce the coefficient of the complexity and the plots of the complexity.
- **distcompute.m**: script that produce the distances between the original decomposition and the starting decomposition.

8.1.3 Tensorlab functions and other used functions

- **btd_nls.m**: the function corresponding to the *Nls* algorithm.
- **btd_minf.m**: the function corresponding to the *Minf* algorithm.
- **mat2tens.m** and **tens2mat.m**: the functions that fold and unfold the tensors into matrices.
- **vec2tens.m** and **tens2vec.m**: the functions that fold and unfold the tensors into vectors.
- **tmprod.m**: the function that computes the mode- N products.
- **qf.m**: the function that returns the Q factor of a QR factorization with only non-negative elements on the diagonal of the R factor.

Note that the functions and scripts implemented during this thesis shares many similarities. To avoid repetitions, the general comments that are shared for all the functions are put in the function **btd_Als.m**, and the comments that are true for all the scripts are put in the script **testfinal.m**.

Specific comments will be present and depend on the function or the script.

8.2 Pseudo-codes

Algorithm 1: Block term decomposition-method *Als*

input : T, U_0, V_0, W_0, A_0
output: Blocks

% initialization of the stopping criteria

 $R = \# \text{ Blocks ; } \text{iter} = 1 ; \text{iterMax} = 700; \text{prec} = 1\text{e-}10; \text{Tol} = 1\text{e-}12;$
while $\text{Err}_{rel}(\text{iter}) \geq \text{prec} \ \& \ \text{NormGrad} \geq \text{Tol} \ \& \ \text{iter} < \text{iterMax}$ **do**
for $i \leftarrow 1$ **to** R **do**

 % $\text{Core}_{\setminus i}$ is the sum of the terms of the decomposition that will be
subtracted to the main tensor to isolate the i -th term

$$\text{Core}_{\setminus i} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$$

 % solve the system to minimize the cost function for this element of the
decomposition

$$U_i = (A_i \bullet_2 V_i \bullet_3 W_i) \setminus (T - \text{Core}_{\setminus i})$$

% factorization of the factor matrix

$$\underline{U}_i = QR(U_i)$$

for $i \leftarrow 1$ **to** R **do**

$$\text{Core}_{\setminus i} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$$

$$V_i = (A_i \bullet_1 U_i \bullet_3 W_i) \setminus (T - \text{Core}_{\setminus i})$$

$$\underline{V}_i = QR(V_i)$$

for $i \leftarrow 1$ **to** R **do**

$$\text{Core}_{\setminus i} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$$

$$W_i = (A_i \bullet_1 U_i \bullet_2 V_i) \setminus (T - \text{Core}_{\setminus i})$$

$$\underline{W}_i = QR(W_i)$$

for $i \leftarrow 1$ **to** R **do**

$$\text{Core}_{\setminus i} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$$

$$A_i = (T - \text{Core}_{\setminus i}) \bullet_1 (U_i)^\dagger \bullet_2 (V_i)^\dagger \bullet_3 (W_i)^\dagger$$

% computation of the gradients w.r.t. the groups of coordinate

$$\text{Grad}(1, r) = dF/dU_r$$

$$\text{Grad}(2, r) = dF/dV_r$$

$$\text{Grad}(3, r) = dF/dW_r$$

$$\text{Grad}(4, r) = dF/dA_r$$

$$\text{NormGrad} = \text{norm}(\text{Grad});$$

$$\tilde{T} = \sum_{r=1}^R A_r \bullet_1 U_r \bullet_2 V_r \bullet_3 W_r;$$

 $\text{iter}++;$

$$\text{Err}_{rel}(\text{iter}) = \text{norm}(T - \tilde{T})/\text{norm}(T);$$

$$\text{Blocks}_r = \{U_r, V_r, W_r, A_r\};$$

Algorithm 2: Block term decomposition-method *line1*

input : T, U_0, V_0, W_0, A_0
output: Blocks

 $R = \# \text{ Blocks ; } \text{iter} = 1 ; \text{iterMax} = 700; \text{prec} = 1\text{e-}10; \text{Tol} = 1\text{e-}12;$
 $\text{size1, size2, size3} = \text{size}(T)$
while $\text{Err}_{rel}(\text{iter}) \geq \text{prec} \ \& \ \text{NormGrad} \geq \text{Tol} \ \& \ \text{iter} < \text{iterMax}$ **do**

 for $m \leftarrow 1$ **to** size1 **do**

 for $i \leftarrow 1$ **to** R **do**

 $\text{Core}_{\setminus i} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$

 $U_i(m, :) = (A_i \bullet_2 V_i \bullet_3 W_i) \setminus (T - \text{Core}_{\setminus i})(m, :, :)$

 for $n \leftarrow 1$ **to** size2 **do**

 for $i \leftarrow 1$ **to** R **do**

 $\text{Core}_{\setminus i} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$

 $V_i(n, :) = (A_i \bullet_1 U_i \bullet_3 W_i) \setminus (T - \text{Core}_{\setminus i})(:, n, :)$

 for $o \leftarrow 1$ **to** size3 **do**

 for $i \leftarrow 1$ **to** R **do**

 $\text{Core}_{\setminus i} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$

 $W_i(o, :) = (A_i \bullet_1 U_i \bullet_2 V_i) \setminus (T - \text{Core}_{\setminus i})(:, :, o)$

 for $i \leftarrow 1$ **to** R **do**

 $U_i = QR(U_i); V_i = QR(V_i); W_i = QR(W_i);$

 for $i \leftarrow 1$ **to** R **do**

 $\text{Core}_{\setminus i} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$

 $A_i = (T - \text{Core}_{\setminus i}) \bullet_1 (U_i)^\dagger \bullet_2 (V_i)^\dagger \bullet_3 (W_i)^\dagger$

 $\text{Grad}(1, r) = dF/dU_r$

 $\text{Grad}(2, r) = dF/dV_r$

 $\text{Grad}(3, r) = dF/dW_r$

 $\text{Grad}(4, r) = dF/dA_r$

 $\text{NormGrad} = \text{norm}(\text{Grad});$

 $\tilde{T} = \sum_{r=1}^R A_r \bullet_1 U_r \bullet_2 V_r \bullet_3 W_r;$

 $\text{iter}++;$

 $\text{Err}_{rel}(\text{iter}) = \text{norm}(T - \tilde{T})/\text{norm}(T);$
 $\text{Blocks}_r = \{U_r, V_r, W_r, A_r\};$

Algorithm 3: Block term decomposition-method *line2*

input : T, U_0, V_0, W_0, A_0

output: Blocks

$R = \# \text{ Blocks} ; \text{iter} = 1 ; \text{iterMax} = 700; \text{prec} = 1\text{e-}10; \text{Tol} = 1\text{e-}12;$
 $\text{size1, size2, size3} = \text{size}(T)$

while $\text{Err}_{rel}(\text{iter}) \geq \text{prec} \ \& \ \text{NormGrad} \geq \text{Tol} \ \& \ \text{iter} < \text{iterMax}$ **do**

for $m \leftarrow 1$ **to** $\text{size1}; n \leftarrow 1$ **to** $\text{size2}; o \leftarrow 1$ **to** size3 **do**

for $i \leftarrow 1$ **to** R **do**

$\text{Core}_{\setminus i} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$
 $U_i(m, :) = (A_i \bullet_2 V_i \bullet_3 W_i) \setminus (T - \text{Core}_{\setminus i})(m, :, :)$

for $i \leftarrow 1$ **to** R **do**

$\text{Core}_{\setminus i} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$
 $V_i(n, :) = (A_i \bullet_1 U_i \bullet_3 W_i) \setminus (T - \text{Core}_{\setminus i})(:, n, :)$

for $i \leftarrow 1$ **to** R **do**

$\text{Core}_{\setminus i} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$
 $W_i(o, :) = (A_i \bullet_1 U_i \bullet_2 V_i) \setminus (T - \text{Core}_{\setminus i})(:, :, o)$

for $i \leftarrow 1$ **to** R **do**

$U_i = QR(U_i); V_i = QR(V_i); W_i = QR(W_i);$

for $i \leftarrow 1$ **to** R **do**

$\text{Core}_{\setminus i} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$
 $A_i = (T - \text{Core}_{\setminus i}) \bullet_1 (U_i)^\dagger \bullet_2 (V_i)^\dagger \bullet_3 (W_i)^\dagger$

$\text{Grad}(1, r) = dF/dU_r$

$\text{Grad}(2, r) = dF/dV_r$

$\text{Grad}(3, r) = dF/dW_r$

$\text{Grad}(4, r) = dF/dA_r$

$\text{NormGrad} = \text{norm}(\text{Grad});$

$\tilde{T} = \sum_{r=1}^R A_r \bullet_1 U_r \bullet_2 V_r \bullet_3 W_r;$

$\text{iter}++;$

$\text{Err}_{rel}(\text{iter}) = \text{norm}(T - \tilde{T})/\text{norm}(T);$

$\text{Blocks}_r = \{U_r, V_r, W_r, A_r\};$

Algorithm 4: Block term decomposition-method *line3*

input : T, U_0, V_0, W_0, A_0

output: Blocks

$R = \# \text{ Blocks ; } \text{iter} = 1 ; \text{iterMax} = 700; \text{prec} = 1\text{e-}10; \text{Tol} = 1\text{e-}12;$
 $\text{size1, size2, size3} = \text{size}(T)$

while $\text{Err}_{rel}(\text{iter}) \geq \text{prec} \ \& \ \text{NormGrad} \geq \text{Tol} \ \& \ \text{iter} < \text{iterMax}$ **do**

for $m \leftarrow 1$ **to** $\text{size1}; n \leftarrow 1$ **to** $\text{size2}; o \leftarrow 1$ **to** size3 **do**

for $i \leftarrow 1$ **to** R **do**

$$\text{Core}_{\setminus i} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$$

$$U_i(m, :) = (A_i \bullet_2 V_i \bullet_3 W_i) \setminus (T - \text{Core}_{\setminus i})(m, :, :)$$

$$V_i(n, :) = (A_i \bullet_1 U_i \bullet_3 W_i) \setminus (T - \text{Core}_{\setminus i})(:, n, :)$$

$$W_i(o, :) = (A_i \bullet_1 U_i \bullet_2 V_i) \setminus (T - \text{Core}_{\setminus i})(:, :, o)$$

for $i \leftarrow 1$ **to** R **do**

$$U_i = QR(U_i); V_i = QR(V_i); W_i = QR(W_i);$$

for $i \leftarrow 1$ **to** R **do**

$$\text{Core}_{\setminus i} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$$

$$A_i = (T - \text{Core}_{\setminus i}) \bullet_1 (U_i)^\dagger \bullet_2 (V_i)^\dagger \bullet_3 (W_i)^\dagger$$

$$\text{Grad}(1, r) = dF/dU_r$$

$$\text{Grad}(2, r) = dF/dV_r$$

$$\text{Grad}(3, r) = dF/dW_r$$

$$\text{Grad}(4, r) = dF/dA_r$$

$$\text{NormGrad} = \text{norm}(\text{Grad});$$

$$\tilde{T} = \sum_{r=1}^R A_r \bullet_1 U_r \bullet_2 V_r \bullet_3 W_r;$$

$\text{iter}++;$

$$\text{Err}_{rel}(\text{iter}) = \text{norm}(T - \tilde{T})/\text{norm}(T);$$

$\text{Blocks}_r = \{U_r, V_r, W_r, A_r\};$

Algorithm 5: Block term decomposition-method *steep*

input : T, U_0, V_0, W_0, A_0
output: Blocks

 $R = \# \text{ Blocks ; } \text{iter} = 1 ; \text{iterMax} = 700; \text{prec} = 1\text{e-}10; \text{Tol} = 1\text{e-}12;$
while $\text{Err}_{\text{rel}}(\text{iter}) \geq \text{prec} \ \& \ \text{NormGrad} \geq \text{Tol} \ \& \ \text{iter} < \text{iterMax}$ **do**

 % computation of the gradient to order the iteration

 $\text{Grad}(1,r) = dF/dU_r$

 $\text{Grad}(2,r) = dF/dV_r$

 $\text{Grad}(3,r) = dF/dW_r$

 $\text{Grad}(4,r) = dF/dA_r$

 % sorts the gradients in descending order of the norm of the corresponding element

 % they are sorted in descending order to update the elements with the largest norm firsts, hence the -1 in the sort function

 $\text{sortedGrad} = \text{sort}(\text{Grad}(:), -1)$

 % the length of sortedGrad is $4R$ because of the R terms and the 4 elements per term

 for $i \leftarrow 1$ **to** $4R$ **do**

 % verify if the i -th element of sortedGrad is the U element of the r -th term

 if $\text{sortedGrad}(i)$ is U_r **then**

 $\text{Core}_{\setminus r} = \sum_{j \neq r} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$

 $U_r = (A_r \bullet_2 V_r \bullet_3 W_r) \setminus (T - \text{Core}_{\setminus r})$

 $U_r = QR(U_r);$

 % verify if the i -th element of sortedGrad is the V element of the r -th term

 if $\text{sortedGrad}(i)$ is V_r **then**

 $\text{Core}_{\setminus r} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$

 $V_r = (A_r \bullet_1 U_r \bullet_3 W_r) \setminus (T - \text{Core}_{\setminus r})$

 $V_r = QR(V_r);$

 % verify if the i -th element of sortedGrad is the W element of the r -th term

 if $\text{sortedGrad}(i)$ is W_r **then**

 $\text{Core}_{\setminus r} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$

 $W_r = (A_r \bullet_1 U_r \bullet_2 V_r) \setminus (T - \text{Core}_{\setminus r})$

 $W_r = QR(W_r);$

 % verify if the i -th element of sortedGrad is the A element of the r -th term

 if $\text{sortedGrad}(i)$ is A_r **then**

 $\text{Core}_{\setminus r} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$

 $A_r = (T - \text{Core}_{\setminus r}) \bullet_1 (U_r)^\dagger \bullet_2 (V_r)^\dagger \bullet_3 (W_r)^\dagger$

 $\text{NormGrad} = \text{norm}(\text{Grad});$

 $\tilde{T} = \sum_{r=1}^R A_r \bullet_1 U_r \bullet_2 V_r \bullet_3 W_r;$

 $\text{iter}++;$

 $\text{Err}_{\text{rel}}(\text{iter}) = \text{norm}(T - \tilde{T}) / \text{norm}(T);$
 $\text{Blocks}_r = \{U_r, V_r, W_r, A_r\};$

Algorithm 6: Block term decomposition-method *steep2*

input : T, U_0, V_0, W_0, A_0
output: Blocks

 $R = \# \text{ Blocks ; } \text{iter} = 1 ; \text{iterMax} = 700; \text{prec} = 1\text{e-}10; \text{Tol} = 1\text{e-}12;$
 $T_0 = \sum_{r=1}^R A_0 \bullet_1 U_0 \bullet_2 V_0 \bullet_3 W_0;$
while $\text{Err}_{rel}(\text{iter}) \geq \text{prec} \ \& \ \text{NormGrad} \geq \text{Tol} \ \& \ \text{iter} < \text{iterMax}$ **do**

% computation of the order criterion for the iteration

 $\text{Order}(s,r) = \|dF/dX_r\|^2 / 2L_{X,r}$

% sorts the gradients in descending order

 % X is used to replace U, V, W or A and s to replace 1,2,3 or 4 to gain some space

if $\text{iter} > 1$ **then**

 $\text{sortedOrder} = \text{sort}(\text{Order}(:), -1)$

 % for the first iteration, we use the same order as A

 for $i \leftarrow 1$ **to** $4R$ **do**

 if $\text{sortedOrder}(i)$ is U_r **then**

 $\text{Core}_{\setminus r} = \sum_{j \neq r} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$

 $U_r = (A_r \bullet_2 V_r \bullet_3 W_r) \setminus (T - \text{Core}_{\setminus r})$

 $U_r = QR(U_r);$

 if $\text{sortedOrder}(i)$ is V_r **then**

 $\text{Core}_{\setminus r} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$

 $V_r = (A_r \bullet_1 U_r \bullet_3 W_r) \setminus (T - \text{Core}_{\setminus r})$

 $V_r = QR(V_r);$

 if $\text{sortedOrder}(i)$ is W_r **then**

 $\text{Core}_{\setminus r} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$

 $W_r = (A_r \bullet_1 U_r \bullet_2 V_r) \setminus (T - \text{Core}_{\setminus r})$

 $W_r = QR(W_r);$

 if $\text{sortedOrder}(i)$ is A_r **then**

 $\text{Core}_{\setminus r} = \sum_{j \neq i} A_j \bullet_1 U_j \bullet_2 V_j \bullet_3 W_j$

 $A_r = (T - \text{Core}_{\setminus r}) \bullet_1 (U_r)^\dagger \bullet_2 (V_r)^\dagger \bullet_3 (W_r)^\dagger$
if $\text{iter} == 1$ **then**

 % at the end of the first iteration, compute $L_{X,r}$

 $T_1 = \sum_{r=1}^R A_1 \bullet_1 U_1 \bullet_2 V_1 \bullet_3 W_1$

 $\text{cost}_0 = \|T - T_0\|^2; \text{cost}_1 = \|T - T_1\|^2$

 $L_{X,r} = \|dF/dX_r\|^2 / 2 * (\text{cost}_0 - \text{cost}_1)$

 $\text{NormGrad} = \text{norm}(\text{Grad});$

 $\tilde{T} = \sum_{r=1}^R A_r \bullet_1 U_r \bullet_2 V_r \bullet_3 W_r;$

 $\text{iter}++;$

 $\text{Err}_{rel}(\text{iter}) = \text{norm}(T - \tilde{T}) / \text{norm}(T);$
 $\text{Blocks}_r = \{U_r, V_r, W_r, A_r\};$

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl