Fast Alternating LS Algorithms for High Order CANDECOMP/PARAFAC Tensor Factorizations

Anh-Huy Phan, *Member, IEEE*, Petr Tichavský, *Senior Member, IEEE*, and Andrzej Cichocki, *Fellow, IEEE*

***Abstract*—CANDECOMP/PARAFAC (CP) has found numerous applications in wide variety of areas such as in chemometrics, telecommunication, data mining, neuroscience, separated representations. For an order-N tensor, most CP algorithms can be computationally demanding due to computation of gradients which are related to products between tensor unfoldings and Khatri-Rao products of all factor matrices except one. These products have the largest workload in most CP algorithms. In this paper, we propose a fast method to deal with this issue. The method also reduces the extra memory requirements of CP algorithms. As a result, we can accelerate the standard alternating CP algorithms 20–30 times for order-5 and order-6 tensors, and even higher ratios can be obtained for higher order tensors (e.g., $N \geq 10$). The proposed method is more efficient than the state-of-the-art ALS algorithm which operates two modes at a time (ALSo2) in the Eigenvector PLS toolbox, especially for tensors with order $N \geq 5$ and high rank.**

***Index Terms*—ALS, CANDECOMP/PARAFAC (CP), canonical decomposition, gradient, tensor factorization.**

I. Introduction

CANDECOMP/PARAFAC (CP), also coined canonical polyadic decomposition, [1], [2] is a common tensor factorization which has found a wide range of applications. For example, CP was applied to analyze the auditory tones by Carroll and Chang [2], or to vowel-sound data by Harshman [1], or to model fluorescence excitation-emission data by hidden loading components in chemometrics [3]. Applications of CP to sensor array processing and CDMA systems in telecommunications have been developed in [4], [5]. In neuroscience, Field and Graupe [6] extracted topographic components model from event-related potentials data, Mørup *et al.* [7] analyzed EEG data in the time-frequency domain. Deburchgraeve *et al.* developed an automatic method for determination of the seizure location in the neonatal brain [8]. Constantine *et al.* [9] modeled the pressure measurements along the combustion chamber as order-6 tensors corresponding to the flight conditions—Mach number, altitude and angle of attack, and the wall temperatures in the combustor and the turbulence mode. Hackbusch and Khoromskij [10] investigated CP approximation to operators and functions in high dimensions. Other applications of CP are in time-varying EEG spectrum [11], data mining [12]–[14], separated representations for generic functions involved in quantum mechanics or kinetic theory descriptions of materials [15]. The CP decomposition has been commonly used over the years because of its uniqueness under mild conditions [16]–[18].

Since the alternating least squares (ALS) algorithm was proposed [1], [2], there have been intensive research efforts to improve performance and accelerate convergence rate of CP algorithms. A number of particular techniques have been developed such as line search extrapolation methods [1], [19]–[22], compression [23]. Instead of alternating estimation, all-at-once algorithms such as the OPT algorithm [24], the conjugate gradient algorithm for nonnegative CP [25], the PMF3, damped Gauss-Newton (dGN) algorithms [21], [26] and fast dGN [27]–[29] have been studied to deal with problems of a slow convergence of the ALS in some cases. Another approach is to consider the CP decomposition as a joint diagonalization problem [30]–[32].

The above mentioned CP algorithms can speed-up convergence rate, or cope with difficult problems. However, in most of existing CP algorithms, the largest workload is product of tensor unfoldings and all-but-one factors which has not been inadequately considered till now. If a tensor of size $I_1 \times I_2 \times \cdots \times I_N$ is an error tensor of a tensor and its CP approximation, the product expresses the gradient of a least squares cost function with respect to a factor matrix of size $I_n \times R$. Hereafter, we call this product "CP gradient" also referred to as "`matricized tensor times Khatri-Rao product`" (`MTTKRP`) [33], [34]. The CP gradients with respect to all the factors have a high computational cost of order $\mathcal{O}(NRJ_N)$ where $J_N = \prod_{n=1}^{N} I_n$ (see detailed cost in Table I). In addition, mode-n tensor unfoldings with $n = 2, 3, \ldots, N-1$ are also time consuming because they permute the order of data entries. For high order tensors $(N \geq 4)$, the CP gradients may become very computationally demanding. Experimental results show that it might take several hours to several months on standard computers to factorize order-12 tensors consisting of million or billion entries (e.g., a tensor of size $I_1 \times I_2 \times \cdots \times I_N, I_n = 5$) and having rank $R \geq I_n$.

In an effort to handle with the CP gradients (`MTTKRP`) over all modes, Tomasi [36] proposed a computation method for order-4 tensors which operates two modes at a time (ALSo2) and reduces the largest number of multiplications from $4RJ_N$

Manuscript received January 25, 2013; revised April 16, 2013 and May 21, 2013; accepted June 08, 2013. Date of publication June 19, 2013; date of current version August 30, 2013. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Xiao-Ping Zhang. The work of P. Tichavský was supported by the Czech Science Foundation through the project 102/09/1278.

A.-H. Phan is with the Lab for Advanced Brain Signal Processing, Brain Science Institute, RIKEN, Wakoshi 351-0198, Japan (e-mail: han@brain.riken.jp).

P. Tichavský is with Institute of Information Theory and Automation, Prague 182 08, Czechoslovakia (e-mail: tichavsk@utia.cas.cz).

A. Cichocki is with the Lab for Advanced Brain Signal Processing, Brain Science Institute, RIKEN, Wakoshi 351-0198, Japan. He is also with Systems Research Institute, Polish Academy of Sciences, Warsaw, Poland (e-mail: cia@brain.riken.jp).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSP.2013.2269903

TABLE I
COMPARISON OF THE NUMBER OF MULTIPLICATIONS EXECUTED IN METHODS TO COMPUTE $\mathbf{Y}_{(n)}(\bigodot_{k\neq n}\mathbf{A}^{(k)})$

Number of multiplications	Unfoldings
$M_{Alg.1}(n) = R\left(\sum_{k=2}^{n-1} J_k + \frac{1}{I_n}\sum_{k=\max\{3,n+1\}}^{N} J_k + J_N\right)$	$\mathbf{Y}_{(n)}$
$M_{Alg.2}(n) = \begin{cases} R\left(\sum_{k=2}^{n-1} J_k + (1-\delta_{nN})\min(J_n, K_{n-1}) + \frac{1}{J_n}\sum_{k=n+2}^{N} J_k + J_N\right), & n = n^\star, n^\star+1, \\ R\sum_{k=2}^{n+1} J_k, \quad n < n^\star, & \\ R\left(K_{n-2} + (1-\delta_{nN})K_{n-1} + \frac{1}{J_n}\sum_{k=n+2}^{N} J_k\right), & n > n^\star+1 \end{cases}$	$\mathbf{Y}_{(1:n^\star)}$ or $\mathbf{Y}_{(1:n^\star-1)}$, $\mathbf{R}_{(1:n)}^{(r,n+1)}$ and $\mathbf{R}_{(n)}^{(r,n)}$, $\mathbf{L}_{(1)}^{(r,n)}$ and $\mathbf{L}_{(1)}^{(r,n-1)}$

$\delta_{i,j}$: Kronecker delta.

to $2RJ_N$. The generalized method for order-N tensor has been implemented as subroutine `alsstep` in the commercial PLS toolbox [37]. However, this method has not yet optimally reduced the computational cost and space cost. In particular, it becomes less efficient when the rank of the decomposition exceeds the largest dimension for odd order N or the product of the two largest ones for even N.

In this paper, a fast computation method is proposed for one mode and all mode CP gradients (`MTTKRP`). The CP gradient (`MTTKRP`) is calculated using two smaller Khatri-Rao products to avoid rearranging entries of a tensor in a computer as much as possible. Moreover, progressive computation of all-mode CP gradients has been further improved by exploiting common factors between CP gradients. It not only has a lower computational cost of $2R\prod_{n=1}^{N} I_n$, but also reduces memory requirement. For example, for hypercube tensors of size $I_n = I$ for all n, the reduction factor is approximately of $\frac{1}{2}I^{N/2-1}$ (details given in Table II). As a result, we formulated the FastALS algorithm which is 20–30 times faster than the ordinary ALS algorithm for order-5, order-6 tensors, and achieves much higher ratios for higher order tensors ($N \geq 10$). The proposed method is also faster and less memory demanding than the ALSo2 algorithm [36], [37], especially in decomposition of tensors of order $N \geq 5$ and high rank.

The paper is organized as follows. Notation and basic multilinear algebra are briefly reviewed in Section II. CP model and CP gradients are shortly reviewed in this section. The fast computation method is presented in Section III. The fast implementation of the ALS algorithm utilizing the fast CP gradient is introduced in Section IV. Section V compares the related ALS algorithm, which is ALSo2, [36], [37] with the proposed algorithm. In Section VI we provide examples illustrating the validity and high performance of the proposed algorithm. Section VII concludes the paper.

II. NOTATION AND CANDECOMP/PARAFAC (CP) MODEL

We shall denote tensors by bold calligraphic letters, e.g., $\mathcal{A} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$, matrices by bold capital letters, e.g., $\mathbf{A} = [\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_R] \in \mathbb{R}^{I\times R}$, and vectors by bold italic letters, e.g., $\boldsymbol{a}_j$ or $\boldsymbol{I} = [I_1, I_2, \ldots, I_N]$.

TABLE II
COMPARISON OF EXTRA MEMORY REQUIREMENTS IN THE CP_ALS ALGORITHMS WHEN UPDATING $\mathbf{A}^{(n)}$ WITHOUT COUNTING SPACE OF THE DATA TENSOR AND FACTORS

	CP_ALS	FastALS
Memory cells	$RJ_{n-1}K_n$	$\begin{cases} R(K_n + J_n), & n = n^\star, n^\star+1, \\ R(J_{n-1} + J_n), & n < n^\star, \\ R(K_{n-1} + K_n), & n > n^\star+1. \end{cases}$
Peak memory cells	RK_1	$R(K_{n^\star} + J_{n^\star})$

An $\boldsymbol{i} = [i_1, i_2, \ldots, i_N]$-th entry $y_{\boldsymbol{i}} = \mathcal{Y}(i_1, i_2, \ldots, i_N)$ with $1 \leq i_n \leq I_n$, $n = 1, 2, \ldots, N$, is alternatively denoted by y_i with the index $i = \text{ivec}(\boldsymbol{i}, \boldsymbol{I})$[1] defined as

$$i = \text{ivec}(\boldsymbol{i}, \boldsymbol{I}) = i_1 + \sum_{n=2}^{N}(i_n - 1)J_{n-1},$$

where $J_n = \prod_{j=1}^{n} I_j$. We also denote $K_n = \prod_{k=n+1}^{N} I_k$, $K_0 = J_N$ and $K_N = 1$. A vector of integer numbers is denoted by colon notation such as $\boldsymbol{k} = i:j = [i, i+1, \ldots, j-1, j]$. For example, we denote $1:n = [1, 2, \ldots, n]$.

Generally, we adopt notation used in [38], [39]. The Kronecker product, the Khatri-Rao (column-wise Kronecker) product, and the (element-wise) Hadamard product are denoted respectively by $\otimes, \odot, \circledast$ [38], [39].

Notation 2.1 (Hadamard and Khatri-Rao products): Given a set of N matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n\times R}$, $n = 1, 2, \ldots, N$, Hadamard and Khatri-Rao products among them are denoted by

$$\mathop{\circledast}_{k\neq n} \mathbf{A}^{(k)} \triangleq \mathbf{A}^{(N)} \circledast \cdots \circledast \mathbf{A}^{(n+1)} \circledast \mathbf{A}^{(n-1)} \circledast \cdots \circledast \mathbf{A}^{(1)},$$

$I_n = I$ for all n,

$$\mathop{\odot}_{n=1}^{N} \mathbf{A}^{(n)} \triangleq \mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n)} \odot \cdots \odot \mathbf{A}^{(1)} \qquad \text{for all } n,$$

$$\mathop{\odot}_{k\neq n} \mathbf{A}^{(k)} \triangleq \mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)}.$$

Definition 2.1 (Reshaping): The reshape[2] operator for a tensor $\mathcal{Y} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$ to a size specified by a vector $\boldsymbol{L} = [L_1, L_2, \ldots, L_M]$ with $\prod_{m=1}^{M} L_m = \prod_{n=1}^{N} I_n$ returns an order-M tensor $\mathcal{X}$, such that $\text{vec}(\mathcal{Y}) = \text{vec}(\mathcal{X})$, and is expressed as

$$\mathcal{X} = \texttt{reshape}(\mathcal{Y}, \boldsymbol{L}) \quad \in \mathbb{R}^{L_1\times L_2\times\cdots\times L_M}. \tag{1}$$

Reshape does not permute entries in its vectorization.

Definition 2.2 (Tensor Unfolding [35], [36], [40]): Unfolding a tensor $\mathcal{Y} \in \mathbb{R}^{I_1\times I_2\times\cdots\times I_N}$ along modes $\boldsymbol{r} = [r_1, r_2, \ldots, r_M]$ and $\boldsymbol{c} = [c_1, c_2, \ldots, c_{N-M}]$ where $[\boldsymbol{r}, \boldsymbol{c}]$ is a permutation of $[1, 2, \ldots, N]$ aims to rearrange this tensor to be a matrix $\mathbf{Y}_{\boldsymbol{r}\times\boldsymbol{c}}$ of size $\prod_{k=1}^{M} I_{r_k} \times \prod_{l=1}^{N-M} I_{c_l}$ whose entries (j_1, j_2) are given by $\mathbf{Y}_{\boldsymbol{r}\times\boldsymbol{c}}(j_1, j_2) = \mathcal{Y}(\boldsymbol{i})$, where

[1] ivec is the "sub2ind" Matlab subroutine.

[2] see the "reshape" Matlab subroutine.

$\boldsymbol{i} = [i_1, \ldots, i_N]$, $\boldsymbol{i_r} = [i_{r_1} \ldots i_{r_M}]$, $\boldsymbol{i_c} = [i_{c_1} \ldots i_{c_{N-M}}]$, $j_1 = \text{ivec}(\boldsymbol{i_r}, \boldsymbol{I_r})$, $j_2 = \text{ivec}(\boldsymbol{i_c}, \boldsymbol{I_c})$.

Remark 2.1:

1) If $\boldsymbol{c} = [c_1 < c_2 < \cdots < c_{N-M}]$, $\mathbf{Y}_{\boldsymbol{r}\times\boldsymbol{c}}$ is simplified to $\mathbf{Y}_{(\boldsymbol{r})}$.
2) If $r = n$ and $\boldsymbol{c} = [1, 2, \ldots, n-1, n+1, \ldots, N]$, we have mode-$n$ matricization $\mathbf{Y}_{\boldsymbol{r}\times\boldsymbol{c}} = \mathbf{Y}_{(n)}$.
3) $\mathbf{Y}_{\boldsymbol{r}\times\boldsymbol{c}} = \mathbf{Y}^T_{\boldsymbol{c}\times\boldsymbol{r}}$.
4) For $\boldsymbol{r} = [1, 2, \ldots, n]$, $\boldsymbol{c} = [n+1, n+2, \ldots, N]$, for all n, $\mathbf{Y}_{\boldsymbol{r}\times\boldsymbol{c}} = \mathbf{Y}_{(\boldsymbol{r})} = \mathbf{Y}_{(1:n)}$ can be expressed and efficiently performed by `reshape`, that is

$$\mathbf{Y}_{(\boldsymbol{r})} \overset{\Delta}{=} \texttt{reshape}\left(\boldsymbol{\mathcal{Y}}, [J_n, K_n]\right). \tag{2}$$

Definition 2.3 (Mode-n Tensor-Vector Product [39]: The mode-n multiplication of a tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ by a vector $\boldsymbol{a} \in \mathbb{R}^{I_n}$ returns an order-$(N-1)$ tensor $\boldsymbol{\mathcal{Z}}$ defined as

$$\text{vec}(\boldsymbol{\mathcal{Z}}) = \mathbf{Y}^T_{(n)}\boldsymbol{a}. \tag{3}$$

Symbolically, the product is denoted by

$$\boldsymbol{\mathcal{Z}} = \boldsymbol{\mathcal{Y}} \bar{\times}_n \boldsymbol{a} \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N}. \tag{4}$$

Tensor-vector product of a tensor $\boldsymbol{\mathcal{Y}}$ with a set of N column vectors $\{\boldsymbol{a}\} = \{\boldsymbol{a}^{(1)}, \boldsymbol{a}^{(2)}, \ldots, \boldsymbol{a}^{(N)}\}$ is denoted by

$$\boldsymbol{\mathcal{Y}} \bar{\times} \{\boldsymbol{a}\} \overset{\Delta}{=} \boldsymbol{\mathcal{Y}} \bar{\times}_1 \boldsymbol{a}^{(1)} \bar{\times}_2 \boldsymbol{a}^{(2)} \cdots \bar{\times}_N \boldsymbol{a}^{(N)}. \tag{5}$$

Definition 2.4 (CANDECOMP/PARAFAC (CP)): CP decomposition means an approximation of a given N-th order tensor by a rank-R tensor of the form

$$\boldsymbol{\mathcal{Y}} \approx \sum_{r=1}^{R} \boldsymbol{a}_r^{(1)} \circ \boldsymbol{a}_r^{(2)} \circ \ldots \circ \boldsymbol{a}_r^{(N)} = \hat{\boldsymbol{\mathcal{Y}}}, \tag{6}$$

where symbol "$\circ$" denotes the outer product, component matrices (factors): $\mathbf{A}^{(n)} = [\boldsymbol{a}_1^{(n)}, \boldsymbol{a}_2^{(n)}, \ldots, \boldsymbol{a}_R^{(n)}] \in \mathbb{R}^{I_n \times R}$, $(n = 1, 2, \ldots, N)$ represent the common (loading) factors [1], [2], [41].

A. Complexity of Tensor Unfoldings

Tensor unfoldings are to rearrange entries of tensors to be matrices. We note that entries of the tensor $\boldsymbol{\mathcal{Y}}$ are stored as a long vector $\text{vec}(\boldsymbol{\mathcal{Y}})$ of the size $J_N = \prod_{n=1}^{N} I_n$ in memory. From this view point, tensor unfolding is to change the order to entries in its vectorization. The more the changes of entries take place, the slower the unfoldings are. Moreover, reading data (entries) stored in non-contiguous blocks will be at a slower rate than accessing data stored in a contiguous block.

The mode-1 unfolding $\mathbf{Y}_{(1)}$ comprises $K_1 = I_2 I_3 \cdots I_N$ column vectors which consist of I_1 contiguous entries of $\boldsymbol{\mathcal{Y}}$, i.e.,

$$\mathbf{Y}_{(1)} = \begin{bmatrix} y_1 & y_{I_1+1} & \cdots & y_{(K_1-1)I_1+1} \\ y_2 & y_{I_1+2} & \cdots & y_{(K_1-1)I_1+2} \\ \vdots & \vdots & \ddots & \vdots \\ y_{I_1} & y_{2I_1} & \cdots & y_{J_N} \end{bmatrix} = [\boldsymbol{\mathcal{Y}}(1\!:\!I_1)\ \boldsymbol{\mathcal{Y}}(I_1\!+\!1\!:\!2I_1)\ \cdots\ \boldsymbol{\mathcal{Y}}((K_1\!-\!1)I_1\!+\!1\!:\!J_N)].$$

By taking into account that $\mathbf{Y}_{(N)} = \mathbf{Y}^T_{(1:N-1)}$, in practice, we compute $\mathbf{Y}_{(1:N-1)}$ instead of $\mathbf{Y}_{(N)}$. $\mathbf{Y}_{(1:N-1)}$ consists of I_N vectors each of which comprises J_{N-1} contiguous entries given by

$$\mathbf{Y}^T_{(N)} = [\boldsymbol{\mathcal{Y}}(1 : J_{N-1}), \boldsymbol{\mathcal{Y}}(J_{N-1} + 1 : 2J_{N-1}), \ldots, \boldsymbol{\mathcal{Y}}((I_N - 1)J_{N-1} + 1 : J_N)].$$

In general, unfoldings $\mathbf{Y}_{(1:n)}$ $(n = 1, 2, \ldots, N)$ do not change the order of entries of $\boldsymbol{\mathcal{Y}}$

$$\mathbf{Y}_{(1:n)} = [\boldsymbol{\mathcal{Y}}(1\!:\!J_n), \boldsymbol{\mathcal{Y}}(J_n\!+\!1\!:\!2J_n), \ldots, \boldsymbol{\mathcal{Y}}((K_n\!-\!1)J_n\!+\!1\!:\!J_N)].$$

Hence, they are relatively fast. We denote by $\boldsymbol{\mathcal{Y}}^{(\boldsymbol{m})}$, $\boldsymbol{m} = [i_{n+1}, i_{n+2}, \ldots, i_N]$, order-$n$ subtensors of $\boldsymbol{\mathcal{Y}}$ whose each entry is given by $\boldsymbol{\mathcal{Y}}^{(\boldsymbol{m})}(i_1, i_2, \ldots, i_n) = \boldsymbol{\mathcal{Y}}(i_1, i_2, \ldots, i_n, i_{n+1}, \ldots, i_N)$. The mode-$n$ unfolding $\mathbf{Y}_{(n)}$ is of the size $I_n \times (J_{n-1}K_n)$, and can be expressed as concatenation of K_n mode-n unfoldings of $\boldsymbol{\mathcal{Y}}^{(\boldsymbol{m})}$, that is

$$\mathbf{Y}_{(n)} = \begin{bmatrix} \mathbf{Y}^{(\boldsymbol{1})}_{(n)} & \cdots & \mathbf{Y}^{(\boldsymbol{m})}_{(n)} & \cdots & \mathbf{Y}^{(\boldsymbol{M})}_{(n)} \end{bmatrix},$$

$$\mathbf{Y}^{(\boldsymbol{m})}_{(n)} = \begin{bmatrix} y_{mJ_n+1} & \cdots & y_{mJ_n+J_{n-1}} \\ y_{mJ_n+J_{n-1}+1} & \cdots & y_{mJ_n+2J_{n-1}} \\ \vdots & \ddots & \vdots \\ y_{mJ_n+(I_n-1)J_{n-1}+1} & \cdots & y_{(m+1)J_n} \end{bmatrix},$$

where $\boldsymbol{M} = [I_{n+1}, I_{n+2}, \ldots, I_N]$ and $m = \text{ivec}(\boldsymbol{m}, \boldsymbol{M}) - 1$. Therefore, most entries of $\mathbf{Y}_{(n)}$ have been permuted. This explains why the mode-n unfoldings $\mathbf{Y}_{(n)}$ for $1 < n < N$ are more time consuming, and relatively slower than unfoldings $\mathbf{Y}_{(1:n)}$.

For example, given a tensor $\boldsymbol{\mathcal{Y}} = \text{reshape}(1 : 24, [2, 3, 4])$ of size $2 \times 3 \times 4$, unfoldings $\mathbf{Y}_{(1)}$, $\mathbf{Y}_{(1:2)} = \mathbf{Y}^T_{(3)}$ and $\mathbf{Y}_{(2)}$ are given by respectively

$$\mathbf{Y}_{(1)} = \begin{bmatrix} 1 & 3 & \cdots & 23 \\ 2 & 4 & \cdots & 24 \end{bmatrix},$$

$$\mathbf{Y}_{(1:2)} = \mathbf{Y}^T_{(3)} = \begin{bmatrix} 1 & 7 & 13 & 19 \\ 2 & 8 & 14 & 20 \\ \vdots & \vdots & \vdots & \vdots \\ 6 & 12 & 18 & 24 \end{bmatrix},$$

$$\mathbf{Y}_{(2)} = \begin{bmatrix} 1 & 2 & \cdots & 19 & 20 \\ 3 & 4 & \cdots & 21 & 22 \\ 5 & 6 & \cdots & 23 & 24 \end{bmatrix}.$$

The permutation of entries of $\mathbf{Y}_{(n)}$ can also be seen from relation between vectorizations $\text{vec}(\mathbf{Y}_{(n)})$ and $\text{vec}(\boldsymbol{\mathcal{Y}})$, which was shown by Tomasi in Paper III in [21]

$$\text{vec}(\boldsymbol{\mathcal{Y}}) = \mathbf{Q}_n\ \text{vec}\left(\mathbf{Y}_{(n)}\right),$$

where $\mathbf{Q}_n$ are commutation matrices of size $J_N \times J_N$, whose explicit expression is given by $\mathbf{Q}_n = \mathbf{I}_{K_n} \otimes \mathbf{P}_{J_{n-1}, I_n}$, where $\mathbf{P}_{I,J}$ is a permutation matrix for any $I \times J$ matrix $\mathbf{X}$ such that $\text{vec}(\mathbf{X}) = \mathbf{P}_{I,J}\text{vec}(\mathbf{X}^T)$ (see Lemma A.1 in [28]). Some other properties of tensor unfolding $\boldsymbol{\mathcal{Y}}_{(n)}$, which are useful to avoid tensor permutation, can be found in [36], and in Section II, Paper

III in [21]. For example, when using an alternative mode-n matricization which arranges $\mathcal{Y}$ to be a matrix $\mathbf{Y}_{(n)}$ of size $I_n \times I_{n+1}I_{n+2}\cdots I_N I_1 \cdots I_{n-1}$, since $\text{vec}(\mathbf{Y}_{(n)}^T) = \text{vec}(\mathbf{Y}_{(n+1)})$, one only needs a transposition and simple reshaping of the data entries [19], [21].

B. Gradients in CP Algorithms

We consider the cost function

$$D = \frac{1}{2}\|\mathcal{Y} - \widehat{\mathcal{Y}}\|_F^2, \tag{7}$$

and the gradients of this cost function with respect to the factors $\mathbf{A}^{(n)}$, $n = 1, 2, \ldots, N$ are given by [21], [28]

$$\begin{aligned}\mathbf{G}^{(n)} &= \mathbf{E}_{(n)}\left(\bigodot_{k\neq n}\mathbf{A}^{(k)}\right)\\ &= \mathbf{Y}_{(n)}\left(\bigodot_{k\neq n}\mathbf{A}^{(k)}\right)\\ &\quad - \mathbf{A}^{(n)}\left(\circledast_{k\neq n}\mathbf{A}^{(k)T}\mathbf{A}^{(k)}\right) \in \mathbb{R}^{I_n\times R},\end{aligned} \tag{8}$$

where $\mathbf{E}_{(n)}$ denote the mode-n unfoldings of the error tensor $\mathcal{E} = \mathcal{Y} - \widehat{\mathcal{Y}}$. The products $\mathbf{E}_{(n)}(\bigodot_{k\neq n}\mathbf{A}^{(k)})$ or $\mathbf{Y}_{(n)}(\bigodot_{k\neq n}\mathbf{A}^{(k)})$ have a computational cost of $\mathcal{O}(RJ_N)$ (see details in Table I), and are the most expensive steps in CP algorithms. Indeed, the mode-n unfoldings $\mathbf{Y}_{(n)}$, for $n > 1$, are time-consuming. The latter products $\mathbf{Y}_{(n)}(\bigodot_{k\neq n}\mathbf{A}^{(k)})$ are more efficient than $\mathbf{E}_{(n)}(\bigodot_{k\neq n}\mathbf{A}^{(k)})$ in the sense of computation because we don't need to construct the error tensor $\mathcal{E}$. However, since both products involve the same mathematical expression, we also call $\mathbf{Y}_{(n)}(\bigodot_{k\neq n}\mathbf{A}^{(k)})$ the CP gradient in which $\mathcal{Y}$ is considered as an error tensor. The product is referred to as `MTTKRP` in the Matlab Tensor toolbox [33], [34].

The CP gradients are employed in almost all CP algorithms. For example, the alternating least squares (ALS) algorithm [1], [2], [5], [19], [42] alternatively minimizes the cost function (7) with an update rule given by

$$\mathbf{A}^{(n)} \leftarrow \mathbf{Y}_{(n)}\left(\bigodot_{k\neq n}\mathbf{A}^{(k)}\right)\left(\circledast_{k\neq n}\mathbf{A}^{(k)T}\mathbf{A}^{(k)}\right)^{\dagger}, \quad \times(n = 1, 2, \ldots, N), \tag{9}$$

where "†" denotes the pseudo-inverse. A fast implementation of ALS for 3-way tensor, the ALSo2 algorithm, [21] reduces the expensive computation of $\mathbf{Y}_{(n)}(\bigodot_{k\neq n}\mathbf{A}^{(k)})$. The method was later extended to higher order decomposition in the PLS toolbox [37]. See Section V for the ALSo2 algorithm.

The all-at-once algorithms such as OPT [24], PMF3, the damped Gauss-Newton (dGN) or fast Levenberg-Marquardt algorithms [21], [26]–[29], [43], [44], the well-known multiplicative algorithm [38], [45] also compute gradients in their update rules.

The direct computation of $\mathbf{Y}_{(n)}(\bigodot_{k\neq n}\mathbf{A}^{(k)})$ for single mode is illustrated in Algorithm 1, and is implemented in the `MTTKRP` function of the Matlab Tensor toolbox [33], [34].

Algorithm 1: Direct Computation of $\mathbf{Y}_{(n)}(\bigodot_{k\neq n}\mathbf{A}^{(k)})$—`MTTKRP` [33], [34]

Input: Data tensor $\mathcal{Y} : (I_1 \times I_2 \times \cdots \times I_N)$, and N factor matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n\times R}$

Output: $\mathbf{G}^{(n)} = \mathbf{Y}_{(n)}(\bigodot_{k\neq n}\mathbf{A}^{(k)}) : I_n \times R$

begin

1 $\quad\mathcal{Y} \leftarrow \texttt{permute}(\mathcal{Y}, [n, 1 : n-1, n+1 : N])$

2 $\quad\mathbf{Y}_{(n)} \leftarrow \texttt{reshape}(\mathcal{Y}, [I_n, J_{n-1}K_n])$

3 $\quad\mathbf{G}^{(n)} = \mathbf{Y}_{(n)}(\bigodot_{k\neq n}\mathbf{A}^{(k)})$

III. Fast Computation of CP Gradient

A. Order of Dimensions

The CP gradient $\mathbf{G}^{(n)} = [\boldsymbol{g}_1^{(n)}, \boldsymbol{g}_2^{(n)}, \ldots, \boldsymbol{g}_R^{(n)}] \in \mathbb{R}^{I_n\times R}$ given by

$$\begin{aligned}\mathbf{G}^{(n)} &= \mathbf{Y}_{(n)}\left(\bigodot_{k\neq n}\mathbf{A}^{(k)}\right)\\ &= \left[\mathbf{Y}_{(n)}\bigotimes_{k\neq n}\boldsymbol{a}_1^{(k)}, \mathbf{Y}_{(n)}\bigotimes_{k\neq n}\boldsymbol{a}_2^{(k)}, \ldots, \mathbf{Y}_{(n)}\bigotimes_{k\neq n}\boldsymbol{a}_R^{(k)}\right]\end{aligned} \tag{10}$$

involves R products

$$\begin{aligned}\boldsymbol{g}_r^{(n)} &= \mathbf{Y}_{(n)}\left(\bigotimes_{k\neq n}\boldsymbol{a}_r^{(k)}\right)\\ &= \mathcal{Y}\bar{\times}_1\boldsymbol{a}_r^{(1)}\cdots\bar{\times}_{n-1}\boldsymbol{a}_r^{(n-1)}\bar{\times}_{n+1}\boldsymbol{a}_r^{(n+1)}\cdots\bar{\times}_N\boldsymbol{a}_r^{(N)}\end{aligned} \tag{11}$$

for $r = 1, 2, \ldots, R$. For $n > 1$, the Kronecker products $\boldsymbol{t} = \bigotimes_{k\neq n}\boldsymbol{a}_r^{(k)}$ can be efficiently computed via the following scheme [33], [34]

$$\begin{aligned}&\boldsymbol{t} \leftarrow \boldsymbol{a}^{(2)} \otimes \boldsymbol{a}^{(1)}, \quad \boldsymbol{t} \leftarrow \boldsymbol{a}^{(3)} \otimes \boldsymbol{t}, \ldots, \boldsymbol{t} \leftarrow \boldsymbol{a}^{(n-1)} \otimes \boldsymbol{t},\\ &\boldsymbol{t} \leftarrow \boldsymbol{a}^{(n+1)} \otimes \boldsymbol{t}, \quad \boldsymbol{t} \leftarrow \boldsymbol{a}^{(n+2)} \otimes \boldsymbol{t}, \ldots, \boldsymbol{t} \leftarrow \boldsymbol{a}^{(N)} \otimes \boldsymbol{t},\end{aligned} \tag{12}$$

with a computational cost of $(\sum_{k=2}^{n-1} J_k + \frac{1}{I_n}\sum_{k=n+1}^{N} J_k)$. When $n = 1$, the cost is given by $\left(\frac{1}{I_1}\sum_{k=3}^{N} J_k\right)$.

For the least cost to compute $\mathbf{G}^{(n)}$, the dimensions of $\mathcal{Y}$ should be in ascending order, i.e., $I_1 \leq I_2 \leq \cdots \leq I_N$. Note that $\boldsymbol{t} = \bigotimes_{k\neq n}\boldsymbol{a}_r^{(k)}$ can be computed from left-to-right, i.e., first computing $\boldsymbol{t} \leftarrow \boldsymbol{a}^{(N)} \otimes \boldsymbol{a}^{(N-1)}$, then $\boldsymbol{t} \leftarrow \boldsymbol{t} \otimes \boldsymbol{a}^{(N-2)}, \ldots$. In this case, the dimensions should be in the descending order. Hereafter, we implicitly assume that the tensor has been rearranged in the ascending order of its dimensions. From (12), computation of $\mathbf{G}^{(n)}$ in (10) requires the following number of multiplications

$$M_{Alg.\ 1}(n) = R\left(J_N + \sum_{k=2}^{n-1} J_k + \frac{1}{I_n}\sum_{k=\max\{3,n+1\}}^{N} J_k\right). \tag{13}$$

B. Fast Gradient with Respect to a Specific Factor

The direct computation of $\mathbf{G}^{(n)}$ in (10) involves the tensor unfolding $\mathbf{Y}_{(n)}$ which is relatively slow to obtain for $1 < n < N$, due to permutation of entries. We note that vectors $\boldsymbol{g}_r^{(n)}$ ($r = 1, 2, \ldots, R$) can be expressed in an equivalent form consisting of tensor-vector products ($\boldsymbol{\mathcal{Y}} \bar{\times}_{k=1}^{n-1} \boldsymbol{a}_r^{(k)}$) and ($\boldsymbol{\mathcal{Y}} \bar{\times}_{l=n+1}^{N} \boldsymbol{a}_r^{(l)}$) on the left side and right side of n, that is

$$\mathbf{Y}_{(n)} \left(\bigotimes_{k \neq n} \boldsymbol{a}_r^{(k)} \right) = \left(\boldsymbol{\mathcal{Y}} \bar{\times}_{k=1}^{n-1} \boldsymbol{a}_r^{(k)} \right) \bar{\times}_{l=2}^{N-n+1} \boldsymbol{a}_r^{(l+n-1)}, \ \text{“left} - \text{to} - \text{right"}, \quad (14)$$

or

$$\mathbf{Y}_{(n)} \left(\bigotimes_{k \neq n} \boldsymbol{a}_r^{(k)} \right) = \left(\boldsymbol{\mathcal{Y}} \bar{\times}_{l=n+1}^{N} \boldsymbol{a}_r^{(l)} \right) \bar{\times}_{k=1}^{n-1} \boldsymbol{a}_r^{(k)}, \ \text{“right} - \text{to} - \text{left"}. \quad (15)$$

The outer tensor-vector products in (14) have been re-indexed since ($\boldsymbol{\mathcal{Y}} \bar{\times}_{k=1}^{n-1} \boldsymbol{a}_r^{(k)}$) is of order-$(N - n + 1)$. The inner tensor-vector products $\boldsymbol{\mathcal{L}}^{(r,n)} \triangleq \boldsymbol{\mathcal{Y}} \bar{\times}_{k=1}^{n-1} \boldsymbol{a}_r^{(k)}$ in (14) and $\boldsymbol{\mathcal{R}}^{(r,n)} \triangleq \boldsymbol{\mathcal{Y}} \bar{\times}_{k=n+1}^{N} \boldsymbol{a}_r^{(k)}$ in (15) can be efficiently computed through $\mathbf{Y}_{(1:n-1)}$ and $\mathbf{Y}_{(1:n)}$

$$\boldsymbol{\mathcal{L}}^{(r,n)} = \text{reshape} \left(\mathbf{Y}_{(1:n-1)}^T \left(\bigotimes_{k=1}^{n-1} \boldsymbol{a}_r^{(k)} \right), [I_n, \ldots, I_N] \right), \quad (16)$$

$$\boldsymbol{\mathcal{R}}^{(r,n)} = \text{reshape} \left(\mathbf{Y}_{(1:n)} \left(\bigotimes_{k=n+1}^{N} \boldsymbol{a}_r^{(k)} \right), [I_1, \ldots, I_n] \right), \quad (17)$$

for $r = 1, 2, \ldots, R$. It means that the unfoldings $\mathbf{Y}_{(n)}$ which are time-consuming for $1 < n < N$ are avoided.

Lemma 3.1: Consider an order-N tensor $\boldsymbol{\mathcal{Y}}$ with $I_1 \leq I_2 \leq \cdots \leq I_N$, the right-to-left projections in (15) and the left-to-right projections in (14) require the following number of multiplications

$$M_{LR}(n) = R \left(J_N + \sum_{k=2}^{n-1} J_k + K_{n-1} + \frac{1}{J_n} \sum_{k=n+2}^{N} J_k \right), \quad (18)$$

$$M_{RL}(n) = R \left(J_N + \sum_{k=2}^{n-1} J_k + J_n + \frac{1}{J_n} \sum_{k=n+2}^{N} J_k \right). \quad (19)$$

Remark 3.1: (See proof in the Appendix).

- The right-to-left projections in (15) are less computationally demanding than Algorithm 1.
- For $J_n > K_{n-1}$, the left-to-right projections in (14) are cheaper than the right-to-left projections in (15).

C. Progressive Computation of All Mode CP Gradients

CP algorithms available in the literature compute all $\mathbf{G}^{(n)}$ for $n = 1, 2, \ldots, N$, either sequentially (in alternating algorithms [1], [2], [5], [19], [42], [45], [46]) or simultaneously (as in all-at-once algorithms [21], [24], [26]–[29], line-search [20], [21]). This section will present a fast method to compute the gradients recursively for $n = 1, \ldots, N$.

Note that

$$\boldsymbol{\mathcal{L}}^{(r,n)} = \boldsymbol{\mathcal{Y}} \bar{\times}_{k=1}^{n-1} \boldsymbol{a}_r^{(k)} = \boldsymbol{\mathcal{L}}^{(r,n-1)} \bar{\times}_1 \boldsymbol{a}_r^{(n-1)}, \quad (20)$$

or

$$\text{vec}(\boldsymbol{\mathcal{L}}^{(r,n)}) = \mathbf{L}_{(1)}^{(r,n-1)T} \boldsymbol{a}_r^{(n-1)}. \quad (21)$$

Similarly,

$$\boldsymbol{\mathcal{R}}^{(r,n)} = \boldsymbol{\mathcal{Y}} \bar{\times}_{k=n+1}^{N} \boldsymbol{a}_r^{(k)} = \boldsymbol{\mathcal{R}}^{(r,n+1)} \bar{\times}_{n+1} \boldsymbol{a}_r^{(n+1)}, \quad (22)$$

or

$$\text{vec} \left(\boldsymbol{\mathcal{R}}^{(r,n)} \right) = \mathbf{R}_{(1:n)}^{(r,n+1)} \boldsymbol{a}_r^{(n+1)}. \quad (23)$$

By exploiting relations in (23) and (21), we can quickly derive $\boldsymbol{\mathcal{R}}^{(n)}$ from $\boldsymbol{\mathcal{R}}^{(n+1)}$, or $\boldsymbol{\mathcal{L}}^{(n)}$ from $\boldsymbol{\mathcal{L}}^{(n-1)}$ instead of computing them as in (17) and (16), respectively. The total number of multiplications of the algorithm summarized in Table I indicates that it is lower than that of Algorithm 1.

The proposed algorithm to compute CP gradients over all modes is summarized in Algorithm 2. Gradient $\mathbf{G}^{(n^\star)}$ (or $\mathbf{G}^{(n^\star+1)}$) is first computed where

$$n^\star = \max\{n : J_n \leq K_{n-1}\}.$$

Other gradients $\mathbf{G}^{(n)}$ are then sequentially computed in the following order $n = n^\star - 1, n^\star - 2, \ldots, 1, n^\star + 1, n^\star + 2, \ldots, N$. Computation of $\mathbf{G}^{(n^\star)}$ and $\mathbf{G}^{(n^\star+1)}$ is the most expensive step. It approximately requires the following number of multiplications

$$\begin{aligned} & R \left(2J_N + 2J_{n^\star} + K_{n^\star} + 2 \sum_{k=2}^{n^\star-1} J_k \right. \\ & \left. \quad + \frac{1}{J_{n^\star}} \sum_{k=n^\star+2}^{N} J_k + \frac{1}{J_{n^\star+1}} \sum_{k=n^\star+3}^{N} J_k \right) \\ & \approx R(2J_N + 2J_{n^\star} + 2K_{n^\star}). \end{aligned} \quad (24)$$

It is straightforward to verify that $\frac{N}{2} \leq n^\star < N$. For $N = 3$, the algorithm first computes $\mathbf{G}^{(2)}$, then $\mathbf{G}^{(1)}$ and $\mathbf{G}^{(3)}$, sequentially, with a total cost of $R(2J_N + 3J_2)$. For $N = 4$, $n^\star = 2$ if $I_1 I_2 > I_4$; otherwise, $n^\star = 3$.

In comparison with Algorithm 1, besides the lower number of multiplications, Algorithm 2 avoids unfoldings $\mathbf{Y}_{(n)}$ ($1 < n < N$) which are time consuming. Therefore, the higher the tensor order is, the more significant the computational saving of Algorithm 2 in comparison to Algorithm 1 is.

Taking into account that there are additional common parts in computing two consecutive gradients which are ($\boldsymbol{a}_r^{(N)} \otimes \cdots \otimes \boldsymbol{a}_r^{(n+2)}$) when computing $\boldsymbol{g}_r^{(n)}$ and $\boldsymbol{g}_r^{(n+1)}$ with $n > n^\star$ in the “left-to-right” strategy (14), and ($\boldsymbol{a}_r^{(n-2)} \otimes \cdots \otimes \boldsymbol{a}_r^{(1)}$) when computing $\boldsymbol{g}_r^{(n)}$ and $\boldsymbol{g}_r^{(n-1)}$ with $n \leq n^\star$ in the “right-to-left” strategy (15). The ALS algorithm which operates two modes at a time (ALSo2) in [36], [37] computes ($\mathbf{A}^{(3)} \odot \cdots \odot \mathbf{A}_r^{(N)}$) before computing $\mathbf{G}^{(1)}$ and $\mathbf{G}^{(2)}$ (see more details in Section V). The similar scheme can be applied to FastALS when $n^\star > 3$ or $n^\star \leq N - 3$. For example, we can compute ($\boldsymbol{a}_r^{(N)} \otimes \cdots \otimes \boldsymbol{a}_r^{(n^\star+2)}$) for both $\boldsymbol{g}^{(n^\star)}$ and $\boldsymbol{g}^{(n^\star+1)}$, and ($\boldsymbol{a}_r^{(N)} \otimes \cdots \otimes \boldsymbol{a}_r^{(n^\star+4)}$) for both $\boldsymbol{g}^{(n^\star+2)}$ and $\boldsymbol{g}^{(n^\star+3)}$.

Remark 3.2: The order of dimensions $I_1 \le I_2 \le \cdots \le I_N$ can be arranged to $I_{p_1} \le I_{p_2} \le \cdots \le I_{p_{n^\star}}$ and $I_{p_{n^\star+1}} \le \cdots \le I_{p_N}$ where $\boldsymbol{p}^\star = [p_1, p_2, \ldots, p_N]$ is a permutation of $[1, 2, \ldots, N]$ such that total computational cost of Alg. 2 $T_{n^\star,N}(\boldsymbol{p}^\star) = \sum_{k=1}^{N} M_{Alg.2}(k)$ is minimum. Note that J_n and K_n in $M_{Alg.2}(k)$ are replaced with $J_n^{\langle \boldsymbol{p} \rangle}$ and $K_n^{\langle \boldsymbol{p} \rangle}$ which are similarly defined but calculated from $\{I_{p_1}, I_{p_2}, \cdots, I_{p_N}\}$.

The optimal $n^\star$ and $\boldsymbol{p}^\star$ can be selected among all possible combinations of $\frac{N}{2} \le n < N$ and at most $\binom{N}{n}$ permutations $\boldsymbol{p}$.

For $N = 3$, the optimal selection is $n^\star = 2$ and $\boldsymbol{p}^\star = [1, 2, 3]$. For $N = 4$, there are only two possible values $n^\star = 2$, 3, and the total costs $T_{n^\star,N}(\boldsymbol{p}) = \sum_{n=1}^{N} M_{Alg.2}(n)$ are given by

$$T_{2,4}(\boldsymbol{p}) = R\left(2J_4 + 2J_2^{\langle \boldsymbol{p} \rangle} + 2K_2^{\langle \boldsymbol{p} \rangle} + \min\left(J_2^{\langle \boldsymbol{p} \rangle}, K_1^{\langle \boldsymbol{p} \rangle}\right) + \min\left(J_3^{\langle \boldsymbol{p} \rangle}, K_2^{\langle \boldsymbol{p} \rangle}\right)\right),$$

$$T_{3,4}(\boldsymbol{p}) = R\left(2J_4 + 3J_2^{\langle \boldsymbol{p} \rangle >} + 2J_3^{\langle \boldsymbol{p} \rangle} + \min\left(J_3^{\langle \boldsymbol{p} \rangle}, K_2^{\langle \boldsymbol{p} \rangle}\right)\right).$$

Remark 3.3: For $N = 4$, Alg. 2 attains the lowest cost in dependence on I_1, I_2, I_3 and I_4:

- If $I_4 < I_1\left(I_3 + 1 - \frac{I_3}{I_2}\right)$, then $n^\star = 2$ and $\boldsymbol{p}^\star = [1, 4, 2, 3]$. That is, dimensions should be rearranged as $[I_1, I_4, I_2, I_3]$.
- Otherwise, $n^\star = 3$ and $\boldsymbol{p}^\star = [1, 2, 3, 4]$.

Remark 3.4: For relatively higher order N, the total cost can be approximated by $T_{n^\star,N}(\boldsymbol{p}) \approx R(2J_N + 3J_n + 3K_{n-1})$, when $n^\star < N - 1$ and $T_{N-1,N}(\boldsymbol{p}) \approx R(2J_N + 3J_{n-1} + 3J_n)$.

Remark 3.5: $[1, 2, \ldots, N]$ is the optimal order for $T_{N-1,N}(\boldsymbol{p})$.

Proof: The remark is deduced from $T_{N-1,N}([1 : k-1, k+1 : N, k]) \ge T_{N-1,N}([1 : k, k+2 : N, k+1])$ for $k = 1, \ldots, N-1$. ∎

We do not investigate details on the optimal $n^\star$ and $\boldsymbol{p}^\star$ for higher order $N \ge 5$ because these parameters can be quickly found by a simple code to loop over all possible combinations which are not more than $\sum_{n=\lceil \frac{N}{2} \rceil}^{N-1} \binom{N}{n} = 2^{N-1} - 1 + \frac{1}{2}\binom{N}{N/2}$ for even N, and $2^{N-1} - 1$ for odd N, respectively. For $N = 5$, 6, there are at most 15, 41 combinations, respectively. Although the optimal order of dimensions and update order of factor matrices can further reduce computational cost of Alg. 2 compared with that with $I_1 \le I_2 \le \cdots \le I_N$, they do not significantly reduce execution time of Alg. 2.

IV. Fast ALS Algorithm

This section presents the fast CP_ALS (FastALS) algorithm in which gradients are computed using Algorithm 2. The update rules (9) are sequentially executed after computing gradients $\mathbf{G}^{(n)}$ in Algorithm 2. The fast ALS algorithm first updates $\mathbf{A}^{(n^\star)}$ instead of $\mathbf{A}^{(1)}$, then sequentially updates other factor matrices following the order $n = n^\star - 1, n^\star - 2, \ldots, 1, n^\star + 1, n^\star + 2, \ldots, N$.

The cost of the (fast) ALS algorithm is highly dependent on the computational cost of $\mathbf{G}^{(n)}$ and cost of computing $\mathbf{G}^{(n)}\boldsymbol{\Gamma}_n^\dagger$ where $\boldsymbol{\Gamma}_n = \circledast_{k \ne n} \mathbf{A}^{(k)T}\mathbf{A}^{(k)}$. Evaluation of the cost function (7) is not expensive and it is quickly computed from the last computed gradient $\mathbf{G}^{(N)}$ (or $\mathbf{G}^{(1)}$) without construction of the approximate tensor $\widehat{\boldsymbol{\mathcal{Y}}}$ [33], [47]

$$\begin{aligned}\|\boldsymbol{\mathcal{Y}} - \widehat{\boldsymbol{\mathcal{Y}}}\|_F^2 &= \|\boldsymbol{\mathcal{Y}}\|_F^2 + \|\widehat{\boldsymbol{\mathcal{Y}}}\|_F^2 - 2\sum_{r}^{R} \boldsymbol{\mathcal{Y}} \bar{\times}_{n=1}^{N} \boldsymbol{a}_r^{(n)} \\ &= \|\boldsymbol{\mathcal{Y}}\|_F^2 + \mathbf{1}_R^T \left(\circledast_{n=1}^{N} \mathbf{A}^{(k)T}\mathbf{A}^{(k)}\right)\mathbf{1}_R \\ &\quad - 2\mathrm{tr}\left\{\mathbf{A}^{(N)T}\mathbf{G}^{(N)}\right\}, \end{aligned} \tag{25}$$

where $\mathbf{1}_R$ is a vector of ones, and $\|\boldsymbol{\mathcal{Y}}\|_F^2$ is computed only once or can be neglected. Hence, when $n^\star < N - 1$, the complexity per iteration of FastALS is given by

Algorithm 2: Fast Computation of $\mathbf{Y}_{(n)}(\bigodot_{k \ne n} \mathbf{A}^{(k)})$ Over All Modes

Input: Data tensor $\boldsymbol{\mathcal{Y}} : (I_1 \times I_2 \times \cdots \times I_N)$,
N factor matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$

Output: $\mathbf{G}^{(n)} = \mathbf{Y}_{(n)}(\bigodot_{k \ne n} \mathbf{A}^{(k)}) : I_n \times R, n = 1, 2, \ldots, N$

begin

1 $n^\star = \max\{n : J_n \le K_{n-1}\}$ where $J_n = I_1 I_2 \ldots I_n$, $K_n = I_{n+1} \cdots I_N$

for $n = n^\star, n^\star - 1, \ldots, 1, n^\star + 1, n^\star + 2, \ldots, N$ **do**

if $(n == n^\star)$ **then**

2 $\mathbf{R}_{(n)}^{(n)} = \texttt{reshape}(\boldsymbol{\mathcal{Y}}, [J_n, K_n])(\bigodot_{k=n+1}^{N} \mathbf{A}^{(k)})$

3 $\mathbf{G}^{(n)} = \texttt{cp\_gradient}(\boldsymbol{\mathcal{R}}^{(n)}, \{\mathbf{A}\}, \, 'right')$

else if$(n == n^\star + 1)$ **then**

4 $\mathbf{L}_{(1)}^{(n)} = (\bigodot_{k=1}^{n-1} \mathbf{A}^{(k)})^T \texttt{reshape}(\boldsymbol{\mathcal{Y}}, [J_{n-1}, K_{n-1}])$

5 $\mathbf{G}^{(n)} = \texttt{cp\_gradient}(\boldsymbol{\mathcal{L}}^{(n)}, \{\mathbf{A}\}, \, 'left')$

else if $(n < n^\star)$ **then**

for $r = 1, 2 \ldots, R$ **do** % Compute $\boldsymbol{\mathcal{R}}^{(r,n)}$ as in (23)

6 $\mathrm{vec}(\boldsymbol{\mathcal{R}}^{(r,n)}) \leftarrow \texttt{reshape}(\boldsymbol{\mathcal{R}}^{(r,n+1)}, [J_n, I_{n+1}])\boldsymbol{a}_r^{(n+1)}$

7 $\mathbf{G}^{(n)} = \texttt{cp\_gradient}(\boldsymbol{\mathcal{R}}^{(n)}, \{\mathbf{A}\}, \, 'right')$

else

for $r = 1, 2, \ldots, R$ **do** % Compute $\boldsymbol{\mathcal{L}}^{(r,n)}$ as in (21)

8 $\mathrm{vec}(\boldsymbol{\mathcal{L}}^{(r,n)}) \leftarrow \texttt{reshape}(\boldsymbol{\mathcal{L}}^{(r,n-1)}, [I_{n-1}, K_{n-1}])^T \boldsymbol{a}_r^{(n-1)}$

9 $\mathbf{G}^{(n)} = \texttt{cp\_gradient}(\boldsymbol{\mathcal{L}}^{(n)}, \{\mathbf{A}\}, \, 'left')$

function $\mathbf{G}^{(n)} = \texttt{cp\_gradient}(\boldsymbol{\mathcal{Z}}^{(n)}, \{\mathbf{A}\}, side)$

for $r = 1, 2, \ldots, R$ **do**

switch $side$

10 **case**$'right'$: $\boldsymbol{g}_r^{(n)} = \texttt{reshape}(\boldsymbol{\mathcal{Z}}^{(r,n)}, [J_{n-1}, I_n])^T(\bigotimes_{k=1}^{n-1} \boldsymbol{a}_r^{(k)})$ % $\boldsymbol{\mathcal{Z}}^{(r,n)} \equiv \boldsymbol{\mathcal{R}}^{(r,n)}$ in (31)

11 **case**$'left'$: $\boldsymbol{g}_r^{(n)} = \texttt{reshape}(\boldsymbol{\mathcal{Z}}^{(r,n)}, [I_n, K_n])(\bigotimes_{k=n+1}^{N} \boldsymbol{a}_r^{(k)})$ % $\boldsymbol{\mathcal{Z}}^{(r,n)} \equiv \boldsymbol{\mathcal{L}}^{(r,n)}$ in (32)

$\boldsymbol{\mathcal{R}}^{(n)}(i_1, \ldots, i_n, r) = \boldsymbol{\mathcal{R}}^{(r,n)}(i_1, \ldots, i_n)$ in Step 2

$\boldsymbol{\mathcal{L}}^{(n)}(r, i_n, \ldots, i_N) = \boldsymbol{\mathcal{L}}^{(r,n)}(i_n, \ldots, i_N)$ in Step 4

1) Computing $\mathbf{G}^{(n)}$ $\sum_{n=1}^{N} M_{Alg.2}(n) \approx R(2J_N + 3J_{n^\star} + 3K_{n^\star})$
2) Computing $\mathbf{\Gamma}_n$ $N(N-2)R^2 + R^2T$, $T = \sum_{n=1}^{N} I_n$
3) Computing $\mathbf{G}^{(n)}\mathbf{\Gamma}_n^{\dagger}$ $NR^3 + R^2T$
4) Evaluating (25) $R^2 + RI_N$

Total $\approx R(2J_N + 3J_{n^\star} + 3K_{n^\star}) + R^2(2T + N^2) + NR^3$

When $n^\star = N-1$, the approximate total cost is $R(2J_N + 3J_{n^\star-1} + 3J_{n^\star}) + R^2(2T + N^2) + NR^3$. The last three steps are common in both ordinary and fast ALS algorithms. Computation of all $\mathbf{G}^{(N)}$ in the ordinary ALS algorithm costs

$$\begin{aligned}\sum_{n=1}^{N} M_{Alg.\ 1}(n) &= R\left(NJ_N + \sum_{n=2}^{N-1}(N-n)J_n + \sum_{n=3}^{N}\left(\sum_{k=1}^{n-1}\frac{1}{I_k}\right)J_n\right)\\ &\approx R\left(N + \sum_{n=1}^{N}\frac{1}{I_n}\right)J_N. \end{aligned} \tag{26}$$

Concerning the memory consumption, in order to update the factor $\mathbf{A}^{(n^\star)}$, FastALS first computes the right-side Khatri-Rao product of size $(K_{n^\star} \times R)$, and then the left-side Khatri-Rao product of size $(J_{n^\star-1} \times R)$. A projection matrix $[\mathrm{vec}(\boldsymbol{\mathcal{R}}^{(1,n^\star)}), \ldots, \mathrm{vec}(\boldsymbol{\mathcal{R}}^{(R,n^\star)})]$ of size $(J_{n^\star} \times R)$ is kept to update other factors. Hence, it needs an extra temporary $\mathcal{O}(R(K_{n^\star} + J_{n^\star}))$ memory cells besides those of the tensor and the factors.

For updating factors $\mathbf{A}^{(n)}$, $n = n^\star - 1, \ldots, 3, 2$, FastALS does not need to access the raw tensor. It utilizes the projection matrix obtained when updating $\mathbf{A}^{(n+1)}$, and constructs a new projection matrix $[\mathrm{vec}(\boldsymbol{\mathcal{R}}^{(1,n)}), \ldots, \mathrm{vec}(\boldsymbol{\mathcal{R}}^{(R,n)})]$ of size $(J_n \times R)$. Moreover, the FastALS algorithm computes only the left-side Khatri-Rao product of size $(J_{n-1} \times R)$. Hence, FastALS requires a temporary $\mathcal{O}(R(J_{n-1}+J_n))$ memory cells. Note that updating $\mathbf{A}^{(1)}$ does not need to compute any Khatri-Rao product and tensor unfolding.

Similarly, to update the factor $\mathbf{A}^{(n^\star+1)}$, FastALS algorithm first computes the left-side Khatri-Rao product of size $(J_{n^\star} \times R)$, then the right-side Khatri-Rao product of size $(K_{n^\star+1} \times R)$, and yields a projection matrix $[\mathrm{vec}(\boldsymbol{\mathcal{L}}^{(1,n^\star+1)}), \ldots, \mathrm{vec}(\boldsymbol{\mathcal{L}}^{(R,n^\star+1)})]$ of size $(K_{n^\star} \times R)$. It requires $\mathcal{O}(R(J_{n^\star} + K_{n^\star}))$ memory cells. Updating factors $\mathbf{A}^{(n)}$, for $n = n^\star+2, \ldots, N-1$, requires a temporary memory of order $\mathcal{O}(R(K_{n-1} + K_n))$, while updating $\mathbf{A}^{(N)}$ does not compute any Khatri-Rao product.

The extra temporary memory of FastALS is summarized in Table II. Without taking into account the unfoldings $\mathbf{Y}_{(n)}$ of size $I_n \times J_{n-1}K_n$, the ordinary CP_ALS algorithm requires an extra $\mathcal{O}(RJ_{n-1}K_n)$ memory cells for the Khatri-Rao products of size $(J_{n-1}K_n \times R)$, $(n = 1, 2, \ldots, N)$. Note that

$$\begin{aligned} J_{n-1}K_{n+1}I_{n+1} > (J_{n-1} + K_{n+1})I_{n+1} \geq J_n + K_n,\\ J_n < K_n, \quad n < n^\star,\\ J_n > K_n, \quad n > n^\star. \end{aligned}$$

Therefore, the FastALS algorithm requires much smaller extra temporary memory than CP_ALS. The maximum extra space of FastALS is of $\mathcal{O}(R(J_{n^\star} + K_{n^\star}))$ memory cells as updating $\mathbf{A}^{(n^\star)}$ or $\mathbf{A}^{(n^\star+1)}$. This confirms that FastALS not only has a lower computational cost, but also requires less memory than CP_ALS. Other alternating algorithms for CPD with/without additional constrains such as nonnegativity, orthogonality [38], [45], [46], [48], [49] can be accelerated in a similar way.

V. Related Works

This section compares Algorithm 2 with the algorithm proposed by Tomasi in [36], which operates two modes at a time (ALSo2), and is implemented as the subroutine `alsstep` in the commercial PLS toolbox [37].

The ALSo2 algorithm reduces the largest number of multiplications NRJ_N to $2RJ_N$, but it has not yet optimally processed the order of dimensions I_n which is also important to reduce the computation cost and space cost. Denote $\mathbf{Z}^{(r,n)} = \boldsymbol{\mathcal{L}}^{(r,n)} \bar{\times}_{k=3}^{N-n+1} \boldsymbol{a}_r^{(n+k-1)} \in \mathbb{R}^{I_n \times I_{n+1}}$, ALSo2 [36], [37] operates two modes n and $(n+1)$ at a time

$$\boldsymbol{g}_r^{(n)} = \mathbf{Z}^{(r,n)}\boldsymbol{a}_r^{(n+1)},\ \boldsymbol{g}_r^{(n+1)} = \mathbf{Z}^{(r,n)^T}\boldsymbol{a}_r^{(n)},\ (r = 1, 2, \ldots, R) \tag{27}$$

through the left-side projection as in (16) with

$$\boldsymbol{\mathcal{L}}^{(r,n)} = \boldsymbol{\mathcal{L}}^{(r,n-2)} \bar{\times}_1 \boldsymbol{a}_r^{(n-2)} \bar{\times}_2 \boldsymbol{a}_r^{(n-1)} \in \mathbb{R}^{I_n \times I_{n+1} \times \cdots \times I_N}, \tag{28}$$

where $r = 1, 2, \ldots, R$, $n = 3, 5, 7, \ldots$, and $\boldsymbol{\mathcal{L}}^{(r,1)} = \boldsymbol{\mathcal{Y}}$. The ALSo2 algorithm first computes $\mathbf{G}^{(1)}$ when N is odd or computes $\mathbf{G}^{(1)}$ and $\mathbf{G}^{(2)}$ for even order N.

For example, when $N = 3$, ALSo2 first computes $\mathbf{G}^{(1)} = \mathbf{Y}_{(1)}(\mathbf{A}^{(3)} \odot \mathbf{A}^{(2)})$ with a cost of $(RJ_N + RK_1)$, then computes $\mathbf{Y}_{(1)}^T\mathbf{A}^{(1)}$ with a cost of RJ_N, and $\mathbf{G}^{(2)}$ and $\mathbf{G}^{(3)}$ with a cost of $2RK_1$. The total cost of ALSo2 is of $R(2J_N + 3K_1)$. A minimum cost is achieved when I_2 and I_3 are the two shortest dimensions. That is, ALSo2 {[36], [37] may need a tensor permutation so that $I_1 \geq I_2 \geq I_3$.

For order-4 tensors, ALSo2 sequentially updates factors $\mathbf{A}^{(2)}$ and $\mathbf{A}^{(1)}$, then $\mathbf{A}^{(3)}$ and $\mathbf{A}^{(4)}$, and requires a total cost of $R(2J_N + 3J_2 + 3K_2)$. The optimal order of dimensions for ALSo2 is $[I_1, I_4, I_3, I_2]$ which is similar to that for FastALS when $n^\star = 2$ in Remark 3.3. However, when $n^\star = 3$, e.g., $I_1I_3 \leq I_4$, FastALS updates factors from right to left, and has a lower cost of $R(2J_N + 3J_3 + 3J_2)$ than ALSo2 (see decomposition of $10 \times 10 \times 10 \times 4000$ dimensional tensors in Table III).

For higher order tensors, in general, the matrices $\mathbf{Z}^{(r,n)}$ cost $R\left(K_{n-1} + \frac{1}{J_{n+1}}\sum_{k=n+3}^{N} J_k\right)$ multiplications, and $\boldsymbol{\mathcal{L}}^{(r,n)}$ cost $R(K_{n-3} + I_{n-2}I_{n-1})$ multiplications. Hence, for even orders N (6, 8, ...), the largest workloads are involved in the computation of $\mathbf{Z}^{(r,1)}$, $\boldsymbol{\mathcal{L}}^{(r,3)}$, $\mathbf{Z}^{(r,3)}$, $\boldsymbol{\mathcal{L}}^{(r,5)}$, $\boldsymbol{\mathcal{Z}}^{(r,5)}$ and $\boldsymbol{\mathcal{L}}^{(r,7)}$ with a total cost of

$$\begin{aligned} R\Bigg(K_0 + \frac{1}{J_2}\sum_{k=4}^{N} J_k + K_0 + I_1I_2 + K_2\\ + \frac{1}{J_4}\sum_{k=6}^{N} J_k + K_2 + I_3I_4 + \cdots\Bigg)\\ \approx R(2J_N + 3K_2 + 3K_4). \end{aligned} \tag{29}$$

For odd orders N (5, 7, ...), the computation cost of ALSo2 [36],[37] is approximately of $R(2J_N + 3K_1 + 3K_3)$. The algo-

rithm requires an extra temporary RK_1 or RK_2 memory cells for odd or even order N, respectively. The optimal order of I_n for ALSo2 can be chosen such that K_1+K_3 or K_2+K_4 is minimum. In general, the dimensions should be in the descending order. The optimal mode permutation is performed just once before the FastALS and ALSo2 algorithms are even started.

We note that ALSo2 [36], [37] can be considered as a particular case of FastALS with a general permutation $\boldsymbol{p}$ in Remark 3.3 when $n^\star = N-1$ for $N = 3, 5, 7, \ldots$ and $n^\star = N-2$ for $N = 4, 6, 8, \ldots$. This algorithm updates factor matrices from right to left. Therefore, in general, FastALS always has lower or equal cost as ALSo2 with an optimal tensor permutation.

VI. Simulations

A. Factorization of Synthetic Tensors

Simulations in this section compare execution times of the ALS algorithms: the ordinary CP_ALS algorithm [1], [2], [34], the FastALS in Section IV, the ALSo2 algorithm, i.e., the subroutine `alsstep` which operates two modes at a time in the commercial PLS toolbox [36], [37].

Comparison of execution times (second) per iteration between CP ALS, FastALS and ALSo2 [35], [36] in factorization of random tensors. Execution times per iteration and speed-up ratio between algorithms are shown as indicated in the below sub-table. Computer PC1, used in all scenarios, has 1.8 GHz i7 processor and 4 GB memory. Computer PC2, used in the last scenario, had 96 GB of RAM and two six-core processors X5690@3.47 GHz.

Example 1 [Factorization of Random Tensors]: We decomposed order-N tensors which were randomly generated from normal distribution with zero mean and unit variance in single-precision floating point with different sizes $I_n = I = 10, 20, \ldots$, for all n. Algorithms factorized the same tensors using the same initial values and ran in 20 iterations without any other stopping criterion. Execution times were measured for various ranks R using the stopwatch command: "tic" "toc" of Matlab release 2011a on a computer (PC1) with a 1.8 GHz i7 processor and 4 GB RAM and the Mac OS X 10.7 operating system. In addition, memory usage of algorithms including allocated memory and peak memory was measured using the Matlab profiler.

Speed ratio between CP_ALS and FastALS is defined as the ratio between their execution times per iteration

$$\rho = \frac{Execution\_time_{\text{ALS}}}{Execution\_time_{\text{FastALS}}}. \tag{30}$$

The final results were averaged over at least 20 iterations $\times 10$ runs. Fig. 1 shows how the speed-up ratio per iteration (times) changes for factorizations of order-3 and order-4 tensors with different sizes I and ranks R. Other results are summarized in Table III. The ratios were relatively high for low rank R, and gradually decreased when increasing R. The fact is that the computational cost of FastALS was also affected by rank R, e.g., the term $NR^3 + TR^2$ where $T = \sum_{n=1}^{N} I_n$ as analyzed in Section IV. Another reason is due to the array-based language Matlab which is relatively slow for relatively high R when executing codes relying on "`for`" loops. Memory requirement is

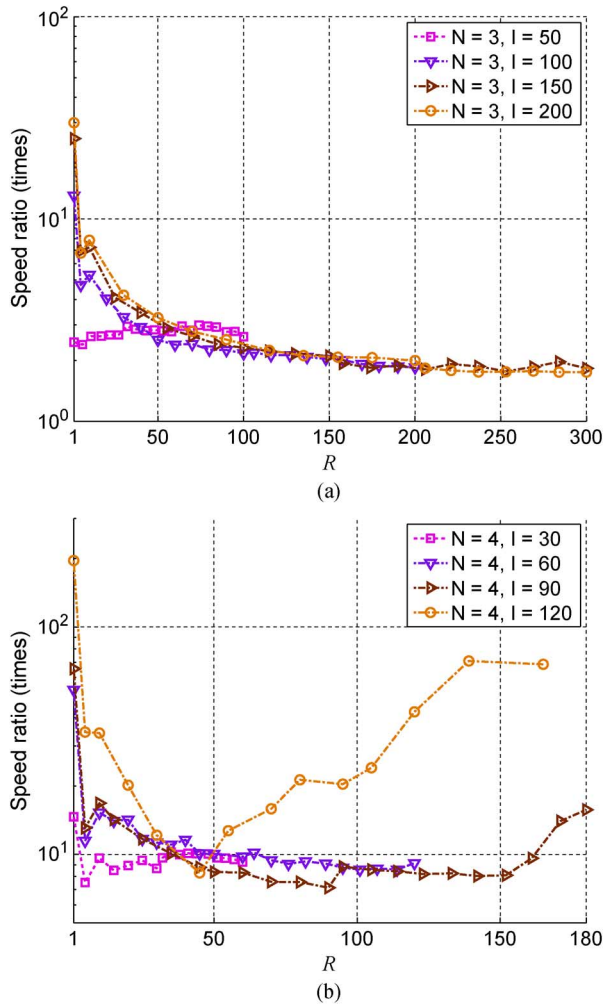

Fig. 1. Speed-up ratios per iteration (in logarithmic scale) for the FastALS algorithm in comparison with the standard CP_ALS algorithm for factorizations of order-3 and order-4 tensors with various sizes $I_n = I$ for all n, and ranks R. (a) Order-3 tensors; (b) order-4 tensors.

also an important issue for the ALS algorithms. The higher the rank-R is, the more memory cells the algorithms require. Since CP_ALS and ALSo2 [37] without proper tensor permutation are much more space-consuming than FastALS, their memory requirements exceed the memory bound sooner than that of FastALS. For this case, the ratio will increase as increasing R. For example, in order to factorize order-4 tensors with $I_n = 120$, for all n, into $R = 50$ rank-one tensors, FastALS allocated 22 MB of physical memory on average per iteration with a peak memory of 5.5 MB for the term $(\bigodot_{k=1}^{n^\star} \mathbf{A}^{(k)})$ where $n^\star = 2$. For the same task, CP_ALS allocated 5.1 GB of memory cells which included the tensor unfoldings $\mathbf{Y}_{(n)}$ and 691 MB of memory for each product $(\bigodot_{k\neq n} \mathbf{A}^{(k)})$, while the computer had only 4 GB of memory. This explains why the speed-up ratio increased when $R \geq 50$ for tensors of size $120\times120\times120\times120$ as illustrated in Fig. 1(b).

Besides the comparison between FastALS and ALS, ALSo2 [37] is also compared with FastALS in Table III. For example, in decompositions of tensors of size $30 \times 100 \times 1000$ and $5 \times 10 \times 500 \times 1000$, FastALS was 4–5 times faster than ALSo2 without optimal tensor permutation. Notice that with a tensor permutation $\boldsymbol{p} = [3, 2, 1]$ or $\boldsymbol{p} = [1, 4, 2, 3]$, ALSo2 is exactly

TABLE III
COMPARISON OF EXECUTION TIMES (SECOND) PER ITERATION BETWEEN CP_ALS, FastALS AND ALSo2 [36], [37] IN FACTORIZATION OF RANDOM TENSORS. EXECUTION TIMES PER ITERATION AND SPEED-UP RATIO BETWEEN ALGORITHMS ARE SHOWN AS INDICATED IN THE BELOW SUB-TABLE. (a) EXPERIMENT RESULTS ON COMPUTER PC1. (b) EXPERIMENT RESULTS ON COMPUTER PC2

(a)

Tensor size	R									
	1		50		100		200		300	
30×100	0.115	0.01	0.228	0.196	0.374	0.397	0.668	1.095	1.017	1.653
×1000	0.006	–	0.052	–	0.103	–	0.235	–	0.328	–
	18	**1**	**4.4**	**1**	**3.6**	**1**	**2.8**	**1**	**3.1**	**1**
	10		20		30		50		100	
5×10	2.83	0.24	6.00	1.04	7.47	0.93	27.4	1.28	196	2.53
×500	0.07	–	0.21	–	0.25	–	0.29	–	0.52	–
×1000	**40**	**1**	**28**	**1**	**30**	**1**	**93**	**1**	**374**	**1**
	20		30		50		100		200	
10×10	0.57	0.05	0.82	0.07	1.16	0.12	2.17	0.37	3.90	0.67
×10×	0.02	0.04	0.03	0.06	0.05	0.11	0.10	0.27	0.22	0.57
4000	**29**	**2**	**27**	**2**	**23**	**2.2**	**22**	**2.7**	**18**	**2.6**
	1		20		30		40		60	
40×40×	5.06	0.17	9.25	2.36	12.7	3.37	28.9	6.25	52.1	38.4
40×40	0.06	0.17	0.46	2.36	0.72	3.37	0.91	6.25	1.06	38.4
×40	**83**	**2.8**	**20**	**5.1**	**18**	**4.7**	**32**	**6.9**	**49**	**36**
	20		50		80		100		150	
30×40×	442	161	–	–	–	–	–	–	–	–
50×60×	1.24	4.77	3.75	96.8	4.28	248	4.71	–	7.34	–
100	**356**	**3.9**	–	**26**	–	**58**	–		–	

(b)

Tensor size	R									
	20		50		80		100		150	
30×40×	17.5	8.11	33.1	19.5	49.7	31.0	60.2	38.6	85.3	56.6
50×60×	0.25	2.66	0.42	6.3	0.57	10.1	0.68	12.3	0.99	18.2
100	**69**	**11**	**80**	**15**	**87**	**18**	**89**	**18**	**86**	**18.4**

(1)	(2)
(3)	(4)
(1)/(3)	(4)/(3)

(1): Execution time of CP_ALS.
(2) and (4): Execution time of ALSo2[36], [37] without and with optimal tensor permutation.
(3): Execution time of FastALS.

FastALS. Table III shows the cases when ALSo2 and FastALS are identical, i.e., the speed ratio is one.

For order-4 tensors of size $10 \times 10 \times 10 \times 4000$, since $n^\star = 3$, FastALS updated factors in the order 3, 2, 1, 4. Table III shows execution times of ALSo2 [37] without and with optimal tensor permutation $\boldsymbol{p} = [1, 4, 2, 3]$. The results indicates that FastALS was 2 times faster than ALSo2 [37] on average when $R \in [20, 200]$. For order-5 tensors of size $I_n = 40$, FastALS was approximately 3–7 times faster than ALSo2 [37] on average for $R \in [1, 40]$. However, when $R = 60$, ALSo2 [37] became very time-consuming. It took 38.25 seconds per iteration on average, and allocated a total 3.6 GB of memory with a peak memory of 1.2 GB. FastALS allocated only a total 121.9 MB of RAM, and took 1 second/iteration. For order-5 tensors of size $30 \times 40 \times 50 \times 60 \times 100$, CP_ALS was nearly impossible to factorize these tensors with rank $R \geq 20$ on PC1. When $R = 20$, with optimal tensor permutation, i.e. [5, 4, 3, 2, 1], the ALSo2 took only 4.77 seconds/iteration, and was 33 times faster than this algorithm without tensor permutation. However, for higher ranks $R \geq 50$, execution times of the ALSo2 dramatically increased because PC1 did not have sufficient memory. For these order-5 tensors, FastALS required only a few seconds per iteration on PC1. Even when running on a computational server (PC2) which had 96 GB of RAM and two six-core Intel Xeon processors X5690@3.47 GHz and the Windows 7 operating system, ALSo2 with optimal tensor permutation was still at least 10 times (up to 18 times) slower than FastALS.

Results for higher order tensors with $N = 10$, 11, 12, 13 and $I_n = 5$ for all n are summarized in Table IV. The execution times were averaged over 30 iterations when $R = 5$, 10, 20 on two computers PC1 and PC2. In addition to speed-up ratios CP_ALS/FastALS and ALSo2/FastALS, the total allocated and peak memory requirements of algorithms are provided. ALSo2 was faster than CP_ALS, but more time consuming than FastALS.

Since $n^\star = 5$, 6 or 7, the products $(\bigodot_{k=1}^{n^\star-1} \mathbf{A}^{(k)})$ and $(\bigodot_{k=n^\star+1}^{N} \mathbf{A}^{(k)})$ occupied not more than 1.562.500 memory cells which in double precision format consumed only 12 MB of memory. For $N = 12$ and $R = 5$, the products $(\bigodot_{k \neq n} \mathbf{A}^{(k)})$ comprised $I^{N-1} \times R = 5^{12} \approx 24410^6$ double-precision numbers which might consume 1.82 GB of RAM. It means that PC1 had insufficient memory for CP_ALS and ALSo2 for $N \geq 12$, and FastALS consumed much less memory than CP_ALS. The speed-up ratio increased as increasing R from 5 to 20. Even when factorizing order-13 tensors, FastALS was still relatively fast and need approximately 3 seconds/iteration.

B. Factorizations of EEG Data

Example 2 [Factorization of Event-Related EEG Time-Frequency Representation]: This example illustrates application of the FastALS algorithm for analysis of real-world EEG data [7], [45] which consists of 28 inter-trial phase coherence (ITPC) measurements [50] of EEG signals of 14 subjects during a proprioceptive pull of the left and right hands. The whole ITPC data set has been represented as a 4-way tensor of $28 \text{ measurements} \times 61 \text{ frequency bins} \times 64 \text{ channels} \times 72 \text{ time frames}$. The first 14 measurements are associated to a group of the left hand stimuli, while the other ones are with the right hand stimuli. Mørup *et al.* analyzed the dataset by nonnegative CP and Tucker components and compared them with components extracted by NMF and ICA [7].

In this example, we approximated the ITPC tensor $\mathcal{Y}$ by CP tensors with various $R = 1, 2, \ldots, 50$. Our aim was to compare the factorization time of CP_ALS and FastALS over various R while interpretation of the results can be found in [7], [45]. Since $n^\star = 2$, ALSo2 and FastALS have identical cost. We only compared CP_ALS and FastALS. Both algorithms used the same initial values and stopped when their differences of successive relative errors $\varepsilon = \frac{\|\mathcal{Y} - \widehat{\mathcal{Y}}\|_F}{\|\mathcal{Y}\|_F}$ were lower than 10^{-6}, or until the maximum number of iterations (2000) was achieved. Factorizations were performed on two computers PC1 and PC2. In addition, the factor matrices in CP_ALS were updated in the same order as in FastALS. This ensured the estimated factors and the numbers of iterations of two algorithms were equivalent up to machine precision.

TABLE IV
COMPARISON OF EXECUTION TIME PER ITERATION AND ALLOCATED MEMORY PER ITERATION BETWEEN CP_ALS, ALSo2 IN [37] AND FastALS IN FACTORIZATIONS OF ORDER-N TENSORS OF SIZE $I_1 = I_2 = \ldots = I_N = 5$, $N = 10, 11, 12, 13$. THE RESULTS WERE MEASURED ON A COMPUTER WHICH HAD 2 × X5690@3.47 GHz CPUs AND 96 GB RAM. VALUES IN PARENTHESES SHOW EXECUTION TIMES ON PC1

		Execution time per iteration (seconds)			Ratio		Allocated Memory (MB)			Peak Memory (MB)		
N	R	FastALS (1)	ALSo2 (2)	CP_ALS (3)	(2)/(1)	(3)/(1)	(1)	(2)	(3)	(1)	(2)	(3)
10	5	0.018 (0.043)	0.12	1.9 (4.1)	7	106	1.2	57	2357	1.0	15	76
	10	0.018 (0.054)	0.20	3.2 (5.5)	11	178	5.1	116	4008	1.2	30	150
	20	0.021	0.38	8.7	18	414	10	229	7313	2.4	60	309
11	5	0.085 (0.188)	2.25 (2.85)	10.8 (20.3)	27	127	3.6	1522	12957	1.1	374	380
	10	0.089 (0.245)	4.39 (5.37)	17.5 (30.8)	49	197	5.2	2952	21997	1.7	783	783
	20	0.098	10.1	43.9	103	448	10	5804	40080	2.5	1566	1566
12	5	0.534 (1)	3.04 (8.91)	59.4 (2257)	6	111	12.6	1522	70647	3.13	392	1959
	10	0.427 (1.171)	5.09 (40)	96.4	12	226	28.9	2953	119767	6.26	786	3917
	20	0.456	14.6	250	32	548	58.0	5804	218006	12.6	1566	7828
13	5	3.71	59.4	331	16	89	12.6	38213	382598	3.13	9785	9813
	10	2.62	123	535	47	204	33.9	73893	647751	6.26	19570	19571
	20	2.76	1533	2491	555	903	67.7	145246	1178081	12.6	39140	39140

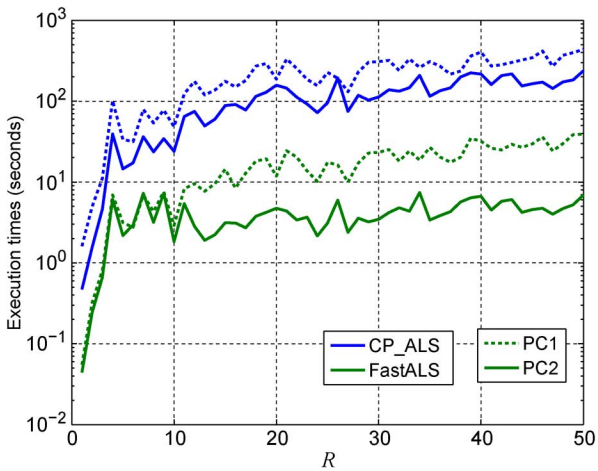

Fig. 2. Execution times (in seconds) of the FastALS algorithm and the standard CP_ALS algorithm in factorization of the order-4 ITPC tensor in Example 2 with various ranks R.

Execution times of the two algorithms illustrated in Fig. 2 indicate that FastALS was faster than CP_ALS on both machines. While CP_ALS required 110–240 seconds to factorize the tensor into $R \geq 30$ components on PC2, FastALS completed the factorizations only in several seconds, and achieved a high speed-up ratio $\rho \approx 25 - 36$ times. The speed ratio on PC1 was around 10–15 times for $R \geq 20$ and lower than that on PC2. *The speed-up ratio depended on the CPU power of the system. We note that PC2 was a computational server with 2 × 3.47 GHz processors each of which had 6 cores, while PC1 was only a laptop with only one Core i7 1.8 GHz*. The execution times of CP_ALS on PC1 were approximately two times longer than those of this algorithm on PC2, while the execution time ratio of FastALS on PC1/PC2 varied from 1 to 7 times as increasing R from 1 to 50. It indicates that FastALS was more efficient on PC2 than PC1.

Example 3 [Factorization of EEG Motor Imagery Data]: In the next set of simulations, we emphasized the superior efficiency of FastALS in comparison to CP_ALS and ALSo2 for factorization of high order tensor which involves left/right motor imagery (MI) movements. We analyzed the EEG MI dataset[3] for the subject 1 which was recorded from 62 channels at a sampling rate of 500 Hz in a duration of 2 seconds per trial. The total number of trials was 200 (100 trials for each class). EEG signals were transformed into the time-frequency domain using the two complex Morlet wavelets CMOR1-1 and CMOR6-1 with the bandwidth parameters $f_b = 1$ Hz and $f_b = 6$ Hz, and the wavelet center frequency $f_c = 1$ Hz [51], giving an order-5 tensor with modes 2 $dictionaries$ × 23 $frequency\ bins$ (8 − 30 Hz)×50 $time\ frames$ × 62 $channels$ × 200 $trials$.

The order-5 tensor was factorized by CP models with various ranks $R = 1, 2, \ldots, 80$. Algorithms were initialized with the same values, and stopped when their differences of successive relative errors $\varepsilon = \frac{\|\mathcal{Y} - \widehat{\mathcal{Y}}\|_F}{\|\mathcal{Y}\|_F}$ were lower than 10^{-6}, or until the maximum number of iterations (2000) was achieved. The order of factor matrices to be updated in CP_ALS was the same as in FastALS so that both algorithms took the same number of iterations. The number of iterations of ALSo2 were set to that of FastALS. In addition to FastALS with $n^\star = 3$ and dimensions in the ascending order, we executed FastALS with the optimal order $\boldsymbol{p} = [1, 3, 4, 2, 5]$. ALSo2 (without reordering dimension) and ALSo2 with a tensor permutation $\boldsymbol{p} = [5, 4, 3, 2, 1]$ were both executed.

Factorization times of algorithms on PC2 are illustrated in Fig. 3. When $R = 20$, FastALS demanded a total 7.6 MB of RAM per iteration with a peak memory of 1.9 MB to allocate the product $\mathbf{A}^{(5)} \odot \mathbf{A}^{(4)}$ comprising $200 \times 62 \times 20 = 248000$ entries in the double precision floating-point type. Both CP_ALS and ALSo2 without optimal tensor permutation computed the Khatri-Rao product $(\bigodot_{n=2}^{5} \mathbf{A}^{(n)})$ which consumed $23 \times 50 \times 62 \times 200 \times 20 \times 8 \approx 2.12$ GB RAM. Moreover, CP_ALS and ALSo2 [36], [37] without a proper tensor permutation allocated a total memory of 5.83 GB and 6.52 GB per iteration respectively, which exceeded the specification of PC1.

With the best tensor permutation $\boldsymbol{p} = [5, 4, 3, 2, 1]$ prior to the factorization, ALSo2 [36], [37] was significantly faster than CP_ALS, but it was still slower than FastALS. The largest Khatri-Rao product computed for the rank-20 case by ALSo2

[3]The data set for single trial EEG classification in BCI is provided by Center for Brain-Like Computing and Machine Intelligence, Shanghai Jiao Tong University, http://bcmi.sjtu.edu.cn/data1/.

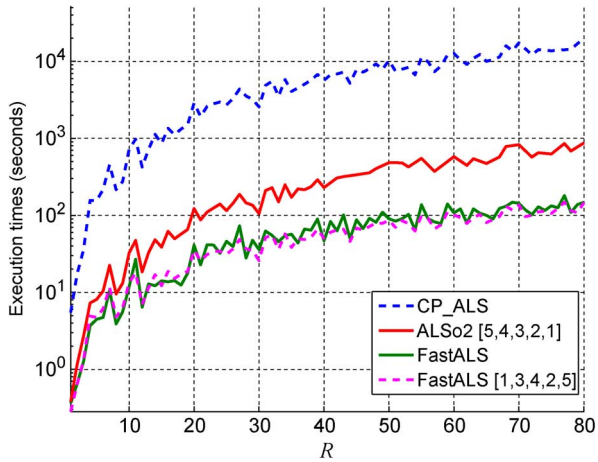

Fig. 3. Execution times (in seconds) of different algorithms in factorization of the order-5 EEG MI tensor in Example 3 as function of R. In the legend, ALSo2 [5, 4, 3, 2, 1] stands for ALSo2 with a tensor permutation $\boldsymbol{p} = [5, 4, 3, 2, 1]$.

only comprised $62 \times 50 \times 23 \times 2 \times 20$ entries, and used 21.8 MB RAM approximately. Fig. 3 indicates that FastALS was approximately 8 times faster than ALSo2 [36], [37] with $\boldsymbol{p} = [5, 4, 3, 2, 1]$ when $R \rightarrow 80$. The FastALS with optimal tensor permutation $\boldsymbol{p} = [1, 3, 4, 2, 5]$ was slightly faster than FastALS with dimensions in the ascending order for high R.

Running on PC2 with a large amount of memory, the CP_ALS took at least 40 minutes to several hours to complete the factorizations, while the FastALS algorithm quickly returned the factors after 1–2 minutes. The big difference in execution times reveals the substantial advantage of the proposed algorithm. The speed up ratio in comparison to CP_ALS was around 100–130 times on PC2, and 300–400 times on PC1 as increasing R to 80. A classification study of MI movements was performed for the same order-5 tensors in [51], [52]. Acceleration of speed in BCI is a key factor because BCI needs to work on-line.

VII. Conclusions

The fast computation of one mode and all mode CP gradients (MTTKRP) has been introduced together with the fast ALS algorithm (FastALS) for the CP decomposition. The proposed method can efficiently and straightforwardly accelerate other alternating algorithms fitting CP in the similar way. For all-at-once optimization algorithms for CPD, CP gradients can be computed sequentially as Algorithm 2, or simultaneously using the method in Section III-B. We show theoretically that the computational cost of FastALS is reduced by a factor of $N/2$ compared with that of the ordinary ALS algorithm, and is smaller than that of ALSo2 [36], [37]. In addition, FastALS demands less memory than ALS with a reduction factor of $\left(\frac{I_1}{J_{n^\star}} + \frac{I_1}{K_{n^\star}}\right)^{-1}$. Therefore, in practice, the speed ratio can be higher than $\frac{N}{2}$, especially on a machine with low memory. We demonstrated that ALSo2 [36], [37] with optimal tensor permutation is equivalent to the FastALS up to for order-4 tensor when $n^\star = 2$ and for higher orders when $n^\star = N - 1$ or $N - 2$. However, FastALS scales far better to higher orders and should therefore be preferred. Finally, FastALS makes standard PCs with relatively low physical memory applicable to factorization of huge and high order tensors. The FastALS algorithm and other alternating and all-at-once algorithms using fast CP gradient are implemented in the Matlab package TENSORBOX which is available online at: http://www.bsp.brain.riken.jp/~phan/tensorbox.php.

Appendix A
Proof of Lemma 3.1

The projection in (15) first computes the order-n tensors $\boldsymbol{\mathcal{R}}^{(r,n)}$ defined in (17) for $r = 1, \ldots, R$, then computes $\boldsymbol{g}_r^{(n)}$ from mode-n unfolding $\mathbf{R}_{(n)}^{(r,n)}$ of $\boldsymbol{\mathcal{R}}^{(r,n)}$

$$\boldsymbol{g}_r^{(n)} = \boldsymbol{\mathcal{R}}_{(n)}^{(r,n)} \left(\bigotimes_{k=1}^{n-1} \boldsymbol{a}_r^{(k)} \right). \tag{31}$$

The two steps require the number of multiplications

$$M_{RL1} = R\left(J_N + \frac{1}{J_n}\sum_{k=n+2}^{N} J_k\right),\ M_{RL2} = R\left(J_n + \sum_{k=2}^{n-1} J_k\right).$$

Hence, the total number of multiplications is given as in (19).

For the left-to-right projection in (14), we build up order-$(N-n+1)$ tensors $\boldsymbol{\mathcal{L}}^{(r,n)}$ of size $I_n \times I_{n+1} \times \cdots \times I_N$ defined in (16), and compute $\boldsymbol{g}_r^{(n)}$ from mode-1 unfolding $\mathbf{L}_{(1)}^{(r,n)}$ of $\boldsymbol{\mathcal{L}}^{(r,n)}$.

$$\boldsymbol{g}_r^{(n)} = \mathbf{L}_{(1)}^{(r,n)} \left(\bigotimes_{k=n+1}^{N} \boldsymbol{a}_r^{(k)} \right). \tag{32}$$

The two steps respectively require the following number of multiplications

$$M_{LR1} = R\left(J_N + \sum_{k=2}^{n-1} J_k\right),\ M_{LR2} = R\left(K_{n-1} + \frac{1}{J_n}\sum_{k=n+2}^{N} J_k\right).$$

Appendix B
Proof of Remark 3.1

Since $I_n \leq I_{n+1}$ and $I_n \leq J_n$, we easily have

$$\begin{aligned} M_{RL}(n) &= R\left(J_N + \frac{1}{J_n}\sum_{k=n+2}^{N} J_k + \sum_{k=2}^{n} J_k\right) \\ &\leq R\left(J_N + \frac{1}{I_n}\sum_{k=n+2}^{N} J_k + J_{n-1}I_{n+1} + \sum_{k=2}^{n-1} J_k\right) \\ &= R\left(J_N + \frac{1}{I_n}\sum_{k=n+1}^{N} J_k + \sum_{k=2}^{n-1} J_k\right) = M_{Alg.\,1}(n), \end{aligned}$$

for $1 < n < N$. It means that the right-to-left projections should be faster than Algorithm 1.

Acknowledgment

The authors would like to thank the referees for the very constructive comments and suggestions which helped to improve

the quality and presentation of the paper. They also thank for the referee who has provided us the Matlab code of the ALSo2 algorithm (the subroutine `alsstep`) [36], [37].

REFERENCES

[1] R. A. Harshman, "Foundations of the PARAFAC procedure: Models and conditions for an explanatory multimodal factor analysis," *UCLA Working Papers in Phonet.* , vol. 16, pp. 1–84, 1970.

[2] J. D. Carroll and J. J. Chang, "Analysis of individual differences in multidimensional scaling via an n-way generalization of Eckart-Young decomposition," *Psychometrika*, vol. 35, no. 3, pp. 283–319, 1970.

[3] C. M. Andersson and R. Bro, "Practical aspects of PARAFAC modelling of fluorescence excitation-emission data," *J. Chemometr.*, vol. 17, pp. 200–215, 2003.

[4] L. De Lathauwer and J. Castaing, "Tensor-based techniques for the blind separation of DS-CDMA signals," *Signal Process.*, vol. 87, no. 2, pp. 322–336, Feb. 2007.

[5] N. D. Sidiropoulos and R. Bro, "PARAFAC techniques for signal separation," in *Signal Processing Advances in Communications*, P. Stoica, G. Giannakis, Y. Hua, and L. Tong, Eds. Upper Saddle River, NJ, USA: Prentice-Hall, 2000, vol. 2, ch. 4.

[6] A. S. Field and D. Graupe, "Topographic component (Parallel Factor) analysis of multichannel evoked potentials: Practical issues in trilinear spatiotemporal decomposition," *Brain Topogr.*, vol. 3, no. 4, pp. 407–423, 1991.

[7] M. Mørup, L. K. Hansen, C. S. Herrmann, J. Parnas, and S. M. Arnfred, "Parallel factor analysis as an exploratory tool for wavelet transformed event-related EEG," *NeuroImage*, vol. 29, no. 3, pp. 938–947, 2006.

[8] W. Deburchgraeve, P. J. Cherian, M. de Vos, R. M. C. Swarte, J. H. Blok, G. H. Visser, P. Govaert, and S. van Huffel, "Neonatal seizure localization using PARAFAC decomposition," *Clin. Neurophysiol.*, vol. 120, no. 10, pp. 1787–1796, Oct. 2009.

[9] P. Constantine, A. Doostan, Q. Wang, and G. Iaccarino, Center for Turbulent Research, "A surrogate accelerated Bayesian inverse analysis of the HyShot II supersonic combustion data," in *Proc. Summer Program*, 2010, AIAA-2011-2037.

[10] W. Hackbusch and B. N. Khoromskij, "Tensor-product approximation to operators and functions in high dimensions," *J. Complex.*, vol. 23, no. 4–6, pp. 697–714, 2007.

[11] H. Becker, P. Comon, L. Albera, M. Haardt, and I. Merlet, "Multi-way space-time-wave-vector analysis for EEG source separation," *Signal Process.*, vol. 92, no. 4, pp. 1021–1031, 2012.

[12] B. W. Bader, M. W. Berry, and M. Browne, "Discussion tracking in Enron email using PARAFAC," in *Survey of Text Mining II*, M. W. Berry and M. Castellanos, Eds. London, U.K.: Springer, 2008, pp. 147–163.

[13] D. M. Dunlavy, T. G. Kolda, and E. Acar, "Temporal link prediction using matrix and tensor factorizations," *ACM Trans. Knowl. Discov. Data*, vol. 5, no. 2, Feb. 2011, pp. Article 10, 27 pages.

[14] A. Shashua and T. Hazan, ICML, "Non-negative tensor factorization with applications to statistics and computer vision," in *Proc. 22th Int. Conf. Mach. Learn. (ICML)*, Bonn, Germany, 2005, pp. 792–799.

[15] D. González, A. Ammar, F. Chinesta, and E. Cueto, "Recent advances on the use of separated representations," *Int. J. Numer. Methods Eng.*, vol. 81, no. 5, pp. 637–659, 2010.

[16] R. A. Harshman, "PARAFAC2: Mathematical and technical notes," *UCLA Working Papers in Phonet.*, vol. 22, pp. 30–44, 1972.

[17] J. B. Kruskal, "Three-way arrays: Rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics," *Linear Algebra and Its Appl.*, vol. 18, no. 2, pp. 95–138, 1977.

[18] N. D. Sidiropoulos and R. Bro, "On the uniqueness of multilinear decomposition of N-way arrays," *J. Chemometr.*, vol. 14, no. 3, pp. 229–239, 2000.

[19] C. A. Andersson and R. Bro, "The N-way toolbox for MATLAB," *Chemometr. Intell. Lab. Syst.*, vol. 52, no. 1, pp. 1–4, 2000.

[20] M. Rajih, P. Comon, and R. A. Harshman, "Enhanced line search: A novel method to accelerate PARAFAC," *SIAM J. Matrix Anal. Appl.*, vol. 30, no. 3, pp. 1128–1147, 2008.

[21] G. Tomasi, "Practical and computational aspects in chemometric data analysis," Ph.D. thesis, Food Science, Quality and Technol., The Royal Veterinary and Agricultural Univ., Frederiksberg, Denmark, 2006.

[22] Y. Chen, D. Han, and L. Qi, "New ALS methods with extrapolating search directions and optimal step size for complex-valued tensor decompositions," *IEEE Trans. Signal Process.*, vol. 59, no. 12, pp. 5888–5898, 2011.

[23] H. A. L. Kiers, "A three-step algorithm for CANDECOMP/PARAFAC analysis of large data sets with multicollinearity," *J. Chemometr.*, vol. 12, no. 3, pp. 155–171, 1998.

[24] E. Acar, D. M. Dunlavy, and T. G. Kolda, "A scalable optimization approach for fitting canonical tensor decompositions," *J. Chemometr.*, vol. 25, no. 2, pp. 67–86, Feb. 2011.

[25] J.-P. Royer, P. Comon, and N. Thirion-Moreau, "Nonnegative 3-way tensor factorization via conjugate gradient with globally optimal stepsize," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process. (ICASSP)*, 2011, pp. 4040–4043.

[26] P. Paatero, "A weighted non-negative least squares algorithm for three-way PARAFAC factor analysis," *Chemometr. Intell. Lab. Syst.*, vol. 38, no. 2, pp. 223–242, 1997.

[27] P. Tichavský and Z. Koldovský, "Simultaneous search for all modes in multilinear models," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process. (ICASSP)*, 2010, pp. 4114–4117.

[28] A.-H. Phan, P. Tichavský, and A. Cichocki, "Low complexity damped Gauss-Newton algorithms for CANDECOMP/PARAFAC," *SIAM J. Matrix Anal. Appl.*, vol. 34, no. 1, pp. 126–147, 2013.

[29] A.-H. Phan, P. Tichavský, and A. Cichocki, "Fast damped Gauss-Newton algorithm for sparse and nonnegative tensor factorization," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process. (ICASSP)*, 2011, pp. 1988–1991.

[30] L. De Lathauwer, "A link between the canonical decomposition in multilinear algebra and simultaneous matrix diagonalization," *SIAM J. Matrix Anal. Appl.*, vol. 28, pp. 642–666, 2006.

[31] F. Roemer and M. Haardt, "A closed-form solution for multilinear PARAFAC decompositions," in *Proc. 5th IEEE Sens. Array Multichannel Signal Process. Workshop (SAM)*, Jul. 2008, pp. 487–491.

[32] L. De Lathauwer and J. Castaing, "Blind identification of underdetermined mixtures by simultaneous matrix diagonalization," *IEEE Trans. Signal Process.*, vol. 56, no. 3, pp. 1096–1105, 2008.

[33] B. W. Bader and T. G. Kolda, "Efficient MATLAB computations with sparse and factored tensors," *SIAM J. Scientif. Comput.*, vol. 30, pp. 205–231, 2007.

[34] B. W. Bader and T. G. Kolda, *MATLAB Tensor Toolbox Version 2.4*. Albuquerque, NM, USA: Sandia National Labs, Jan. 2010 [Online]. Available: http://csmr.ca.sandia.gov/~tgkolda/TensorToolbox/

[35] B. W. Bader and T. G. Kolda, "Algorithm 862: MATLAB tensor classes for fast algorithm prototyping," *ACM Trans. Math. Software*, vol. 32, no. 4, pp. 635–653, 2006.

[36] G. Tomasi, "Recent developments in fast algorithms for fitting the PARAFAC model," in *Proc. TRICAP*, Crete, Greece, 2006 [Online]. Available: http://www.telecom.tuc.gr/nikos/TRICAP2006main/TomasiTRICAP2006.ppt

[37] Eigenvector Research, Inc., PLS Toolbox [Online]. Available: http://www.eigenvector.com/software/-pls_toolbox.htm 2012

[38] A. Cichocki, R. Zdunek, A.-H. Phan, and S. Amari, *Nonnegative Matrix and Tensor Factorizations: Applications to Exploratory Multi-Way Data Analysis and Blind Source Separation*. Chichester, U.K.: Wiley, 2009.

[39] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Rev.*, vol. 51, no. 3, pp. 455–500, Sep. 2009.

[40] S. Ragnarsson and C. F. Van Loan, "Block tensor unfoldings," *SIAM J. Matrix Anal. Appl.*, vol. 33, no. 1, pp. 149–169, 2012.

[41] F. L. Hitchcock, "The expression of a tensor or a polyadic as a sum of products," *J. Math. Phys.*, vol. 6, pp. 164–189, 1927.

[42] A. Smilde, R. Bro, and P. Geladi, *Multi-Way Analysis: Applications in the Chemical Sciences*. New York, NY, USA: Wiley, 2004.

[43] P. Tichavský, A.-H. Phan, and Z. Koldovský, "Cramér-Rao-induced bounds for CANDECOMP/PARAFAC tensor decomposition," *IEEE Trans. Signal Process.*, vol. 61, no. 8, pp. 1986–1997, 2013.

[44] P. Tichavský, A.-H. Phan, and A. Cichocki, "A further improvement of a fast damped GAUSS-NEWTON algorithm for CANDECOMP-PARAFAC tensor decomposition," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process. (ICASSP)*, 2013, pp. 5964–5968.

[45] M. Mørup, L. K. Hansen, and S. M. Arnfred, "ERPWAVELAB a toolbox for multi-channel analysis of time-frequency transformed event related potentials," *J. Neurosci. Methods*, vol. 161, no. 2, pp. 361–368, 2006.
[46] A. Karfoul, L. Albera, and L. De Lathauwer, "Iterative methods for the canonical decomposition of multi-way arrays: Application to blind underdetermined mixture identification," *Signal Process.*, vol. 91, no. 8, pp. 1789–1802, 2011.
[47] G. Tomasi and R. Bro, "2.22—Multilinear models: Iterative methods," in *Comprehensive Chemometrics: Chemical and Biochemical Data Analysis*, S. Brown, R. Tauler, and B. Walczak, Eds. Oxford, U.K.: Elsevier, 2009, pp. 411–451.
[48] A. Cichocki and A.-H. Phan, "Fast local algorithms for large scale nonnegative matrix and tensor factorizations," *IEICE Trans.*, vol. 92-A, no. 3, pp. 708–721, 2009.
[49] J. Chen and Y. Saad, "On the tensor SVD and the optimal low rank orthogonal approximation of tensors," *SIAM J. Matrix Anal. Appl.*, vol. 30, no. 4, pp. 1709–1734, Jan. 2009.
[50] C. Tallon-Baudry, O. Bertrand, C. Delpuech, and J. Pernier, "Stimulus specificity of phase-locked and non-phase-locked 40 Hz visual responses in human," *J. Neurosci.*, vol. 16, no. 13, pp. 4240–4249, 1996.
[51] A.-H. Phan and A. Cichocki, "Tensor decompositions for feature extraction and classification of high dimensional datasets," *IEICE Nonlin. Theory and Its Appl.*, vol. 1, pp. 37–68, 2010, (invited paper).
[52] A.-H. Phan, NFEA: Tensor Toolbox for Feature Extraction and Applications [Online]. Available: http://www.bsp.brain.riken.jp/~phan/nfea.html 2011

Anh-Huy Phan (M'13) received the master degree from Hochiminh University of Technology, Vietnam, in 2005, and the Ph.D. degree from the Kyushu Institute of Technology, Japan, in 2011.

He worked as Deputy head of Research and Development Department, Broadcast Research and Application Center, Vietnam Television, and is currently Research Scientist at the Laboratory for Advanced Brain Signal Processing, and a visiting research scientist in Toyota Collaboration Center, Brain Science Institute, RIKEN. His research interests include multilinear algebra, tensor computation, blind source separation, and brain computer interface.

Petr Tichavský (M'98–SM'04) received the M.S. degree in mathematics in 1987 from the Czech Technical University, Prague, Czechoslovakia, and the Ph.D. degree in theoretical cybernetics from the Czechoslovak Academy of Sciences in 1992.

Since that time he has been with the Institute of Information Theory and Automation, Academy of Sciences of the Czech Republic, Prague. In 1994, he received the *Fulbright grant* for a 10 month fellowship at the Department of Electrical Engineering, Yale University, New Haven, CT. He is author and coauthor of research papers in the area of sinusoidal frequency/frequency-rate estimation, adaptive filtering and tracking of time varying signal parameters, algorithm-independent bounds on achievable performance, sensor array processing, independent component analysis, and blind source separation.

Dr. Tichavský received the *Otto Wichterle Award* from Academy of Sciences of the Czech Republic in 2002. He served as an Associate Editor of the IEEE SIGNAL PROCESSING LETTERS from 2002 to 2004, and as Associate Editor of the IEEE TRANSACTIONS ON SIGNAL PROCESSING from 2005 to 2009 and again from 2011 to now. He has also served as a General Co-Chair of the 36th IEEE International Conference on Acoustics, Speech and Signal Processing ICASSP 2011 in Prague.

Andrzej Cichocki (M'96–SM'06–F'13) received the M.Sc. (with honors), Ph.D., and Dr.Sc. (Habilitation) degrees, all in electrical engineering from Warsaw University of Technology, Poland.

He spent several years at University Erlangen, Germany, as an Alexander-von-Humboldt Research Fellow and Guest Professor. In 1995–1997, he was a team leader of the Laboratory for Artificial Brain Systems, Frontier Research Program RIKEN, Japan, in the Brain Information Processing Group. He is currently a Senior Team Leader and Head of the laboratory for Advanced Brain Signal Processing, at RIKEN Brain Science Institute, Japan, and Professor in IBS, PAN, Poland. His researches focus on tensor decompositions, brain machine interface, human robot interactions, EEG hyper-scanning, brain to brain interface, and their practical applications.

Dr. Cichocki has served as an Associate Editor of the IEEE TRANSACTIONS ON NEURAL NETWORKS, IEEE TRANSACTIONS ON SIGNALS PROCESSING, the *Journal of Neuroscience Methods*, and as founding Editor-in-Chief for the *Journal Computational Intelligence and Neuroscience*. He is (co)author of more than 400 technical journal papers and four monographs in English (two of them translated to Chinese). His publications currently report over 19 000 citations according to Google Scholar, with an h-index of 59.