SANDIA REPORT

SAND2006-7592
Unlimited Release
Printed December 2006

Efficient MATLAB computations with sparse and factored tensors

Brett W. Bader and Tamara G. Kolda

Sandia National Laboratories

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.osti.gov/bridge

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2006-7592
Unlimited Release
Printed December 2006

Efficient MATLAB computations with sparse and factored tensors

Brett W. Bader
Applied Computational Methods Department
Sandia National Laboratories
Albuquerque, NM 87185-0316
`bwbader@sandia.gov`

Tamara G. Kolda
Computational Science and Mathematics Research Department
Sandia National Laboratories
Livermore, CA 94550-9159
`tgkolda@sandia.gov`

Abstract

In this paper, the term tensor refers simply to a multidimensional or N-way array, and we consider how specially structured tensors allow for efficient storage and computation. First, we study sparse tensors, which have the property that the vast majority of the elements are zero. We propose storing sparse tensors using coordinate format and describe the computational efficiency of this scheme for various mathematical operations, including those typical to tensor decomposition algorithms. Second, we study factored tensors, which have the property that they can be assembled from more basic components. We consider two specific types: a Tucker tensor can be expressed as the product of a core tensor (which itself may be dense, sparse, or factored) and a matrix along each mode, and a Kruskal tensor can be expressed as the sum of rank-1 tensors. We are interested in the case where the storage of the components is less than the storage of the full tensor, and we demonstrate that many elementary operations can be computed using only the components. All of the efficiencies described in this paper are implemented in the Tensor Toolbox for MATLAB.

Acknowledgments

We gratefully acknowledge all of those who have influenced the development of the Tensor Toolbox through their conversations and email exchanges with us—you have helped us to make this a much better package. In particular, we thank Evrim Acar, Rasmus Bro, Jerry Gregoire, Richard Harshman, Morten Mørup, and Giorgio Tomasi. We also thank Jimeng Sun for being a beta tester and using the results in [43].

Contents

Tables

1 Introduction

Tensors, by which we mean multidimensional or N-way arrays, are used today in a wide variety of applications, but many issues of computational efficiency have not yet been addressed. In this article, we consider the problem of efficient computations with sparse and factored tensors, whose dense/unfactored equivalents would require too much memory.

Our particular focus is on the computational efficiency of tensor decompositions, which are being used in an increasing variety of fields in science, engineering, and mathematics. Tensor decompositions date back to the late 1960s with work by Tucker [49], Harshman [18], and Carroll and Chang [8]. Recent decades have seen tremendous growth in this area with a focus towards improved algorithms for computing the decompositions [12, 11, 55, 48]. Many innovations in tensor decompositions have been motivated by applications in chemometrics [3, 30, 7, 42]. More recently, these methods have been applied to signal processing [9, 10], image processing [50, 52, 54, 51], data mining [41, 44, 1], and elsewhere [25, 35]. Though this work can be applied in a variety of contexts, we concentrate on operations that are common to tensor decompositions, such as Tucker [49] and CANDECOMP/PARAFAC [8, 18].

For the purposes of our introductory discussion, we consider a third-order tensor

$$\mathcal{X} \in \mathbb{R}^{I \times J \times K}.$$

Storing every entry of $\mathcal{X}$ requires IJK storage. A sparse tensor is one where the overwhelming majority of the entries are zero. Let P denote the number of nonzeros in $\mathcal{X}$. Then, we say $\mathcal{X}$ is sparse if $P \ll IJK$. Typically, only the nonzeros and their indices are stored for a sparse tensor. We discuss several possible storage schemes and select coordinate format as the most suitable for the types of operations required in tensor decompositions. Storing a tensor in coordinate format requires storing P nonzero values and NP corresponding integer indices, for a total of $(N+1)P$ storage.

In addition to sparse tensors, we study two special types of factored tensors that correspond to the Tucker [49] and CANDECOMP/PARAFAC [8, 18] models. Tucker format stores a tensor as the product of a core tensor and a factor matrix along each mode [24]. For example, if $\mathcal{X}$ is a third-order tensor that is stored as the product of a core tensor $\mathcal{G}$ of size $R \times S \times T$ with corresponding factor matrices, then we express it as

$$\mathcal{X} = [\![\mathcal{G} \, ; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!] \quad \text{which means} \quad x_{ijk} = \sum_{r=1}^{R} \sum_{s=1}^{S} \sum_{t=1}^{T} g_{rst} \, a_{ir} b_{js} c_{kt} \text{ for all } i, j, k.$$

If $I, J, K \gg R, S, T$, then forming $\mathcal{X}$ explicitly requires more memory than is needed to store only its components. The storage for the factored form with a dense core tensor is $RST + IR + JS + KT$. However, the Tucker format is not limited to the case where $\mathcal{G}$ is dense and smaller than $\mathcal{X}$. It could be the case that $\mathcal{G}$ is a large, sparse tensor so that $R, S, T \gg I, J, K$ but the total storage is still less than IJK. Thus, more generally, the storage for a Tucker tensor is $\textsc{storage}(\mathcal{G}) + IR + JS + KT$. Kruskal format stores a tensor as the sum of rank-1 tensors [24]. For example, if $\mathcal{X}$ is a third-order tensor that is stored as the sum of R rank-1 tensors, then we express it as

$$\mathcal{X} = [\![\boldsymbol{\lambda}\,;\mathbf{A},\mathbf{B},\mathbf{C}]\!] \quad \text{which means} \quad x_{ijk} = \sum_{r=1}^{R} \lambda_r\, a_{ir} b_{jr} c_{kr} \text{ for all } i, j, k.$$

As with the Tucker format, when $I, J, K \gg R$, forming $\mathcal{X}$ explicitly requires more memory than storing just its factors, which require only $(I + J + K + 1)R$ storage.

These storage formats and the techniques in this article are implemented in the MATLAB Tensor Toolbox, Version 2.1 [5].

1.1 Related Work & Software

MATLAB (Version 2006a) provides dense multidimensional arrays and operations for elementwise and binary operations. Version 1.0 of our MATLAB Tensor Toolbox [4] extends MATLAB's core capabilities to support operations such as tensor multiplication and matricization. The previous version of the toolbox also included objects for storing Tucker and Kruskal factored tensors but did not support mathematical operations on them beyond conversion to unfactored format. MATLAB cannot store sparse tensors except for sparse matrices which are stored in CSC format [15]. Mathematica, an alternative to MATLAB, also supports multidimensional arrays, and there is a Mathematica package for working with tensors that accompanies the book [39]. In terms of sparse arrays, Mathematica stores it `SparseArray`'s in CSR format and claims that its format is general enough to describe arbitrary order tensors.[1] Maple has the capacity to work with sparse tensors using the `array` command and supports mathematical operations for manipulating tensors that arise in the context of physics and general relativity.

There are two well known packages for (dense) tensor decompositions. The N-way toolbox for MATLAB by Andersson and Bro [2] provides a suite of efficient functions and alternating least squares algorithms for decomposing dense tensors into a variety of models including Tucker and CANDECOMP/PARAFAC. The Multilinear Engine by Paatero [36] is a FORTRAN code based on on the conjugate gradient algorithm that also computes a variety of multilinear models. Both packages can handle missing data and constraints (e.g., nonnegativity) on the models.

A few other software packages for tensors are available that do not explicitly target tensor decompositions. A collection of highly optimized, template-based tensor classes in C++ for general relativity applications has been written by Landry [29] and

[1]Visit the Mathematica web site (`www.wolfram.com`) and search on "SparseArray Data Format."

supports functions such as binary operations and internal and external contractions. The tensors are assumed to be dense, though symmetries are exploited to optimize storage. The most closely related work to this article is the HUJI Tensor Library (HTL) by Zass [53], a C++ library for dealing with tensors using templates. HTL includes a `SparseTensor` class that stores index/value pairs using an STL map. HTL addresses the problem of how to optimally sort the elements of the sparse tensor (discussed in more detail in §3.1) by letting the user specify how the subscripts should be sorted. It does not appear that HTL supports general tensor multiplication, but it does support inner product, addition, elementwise multiplication, and more. We also briefly mention MultiArray [14], which provides a general array class template that supports multiarray abstractions and can be used to store dense tensors.

Because it directly informs our proposed data structure, related work on storage formats for sparse matrices and tensors is deferred to section §3.1.

1.2 Outline of article

In §2, we review notation and matrix and tensor operations that are needed in the paper. In §3, we consider sparse tensors, motivate our choice of coordinate format, and describe how to make operations with sparse tensors efficient. In §4, we describe the properties of the Tucker tensor and demonstrate how they can be used for efficient computations. In §5, we do the same for the Kruskal tensor. In §6, we discuss inner products and elementwise multiplication between the different types of tensors. Finally, in §7, we conclude with a discussion on the Tensor Toolbox, our implementation of these concepts in MATLAB.

This page intentionally left blank.

2 Notation and Background

We follow the notation of Kiers [22], except that tensors are denoted by boldface Euler script letters, e.g., $\boldsymbol{\mathscr{X}}$, rather than using underlined boldface $\underline{\mathbf{X}}$. Matrices are denoted by boldface capital letters, e.g., $\mathbf{A}$; vectors are denoted by boldface lowercase letters, e.g., $\mathbf{a}$; and scalars are denoted by lowercase letters, e.g., a. MATLAB-like notation specifies subarrays. For example, let $\boldsymbol{\mathscr{X}}$ be a third-order tensor. Then, $\mathbf{X}_{i::}$, $\mathbf{X}_{:j:}$, and $\mathbf{X}_{::k}$ denote the horizontal, lateral, and frontal slices, respectively. Likewise, $\mathbf{x}_{:jk}$, $\mathbf{x}_{i:k}$, and $\mathbf{x}_{ij:}$ denote the column, row, and tube fibers. A single element is denoted by x_{ijk}. As an exception, provided that there is no possibility for confusion, the rth column of a matrix $\mathbf{A}$ is denoted as $\mathbf{a}_r$. Generally, indices are taken to run from 1 to their capital version, i.e., $i = 1, \ldots, I$. All of the concepts in this section are discussed at greater length in Kolda [24]. For sets we use calligraphic font, e.g., $\mathcal{R} = \{r_1, r_2, \ldots, r_P\}$. We denote a set of indices by $I_{\mathcal{R}} = \{I_{r_1}, I_{r_2}, \ldots, I_{r_P}\}$.

2.1 Standard matrix operations

The Kronecker product of matrices $\mathbf{A} \in \mathbb{R}^{I \times J}$ and $\mathbf{B} \in \mathbb{R}^{K \times L}$ is

$$\mathbf{A} \otimes \mathbf{B} \equiv \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1J}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2J}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}\mathbf{B} & a_{I2}\mathbf{B} & \cdots & a_{IJ}\mathbf{B} \end{bmatrix} \in \mathbb{R}^{IK \times JL}.$$

The Khatri-Rao product [34, 38, 7, 42] of matrices $\mathbf{A} \in \mathbb{R}^{I \times K}$ and $\mathbf{B} \in \mathbb{R}^{J \times K}$ is

$$\mathbf{A} \odot \mathbf{B} \equiv \begin{bmatrix} \mathbf{a}_1 \otimes \mathbf{b}_1 & \mathbf{a}_2 \otimes \mathbf{b}_2 & \cdots & \mathbf{a}_K \otimes \mathbf{b}_K \end{bmatrix} \in \mathbb{R}^{IJ \times K}.$$

The Hadamard (elementwise) product of matrices $\mathbf{A}$ and $\mathbf{B}$ is denoted by $\mathbf{A} * \mathbf{B}$. See, e.g., [42] for properties of these operators.

2.2 Vector outer product

The symbol $\circ$ denotes the vector outer product. Let $\mathbf{a}^{(n)} \in \mathbb{R}^{I_n}$ for all $n = 1, \ldots, N$. Then the outer product of these N vectors is an N-way tensor, defined elementwise as

$$\left(\mathbf{a}^{(1)} \circ \mathbf{a}^{(2)} \circ \cdots \circ \mathbf{a}^{(N)}\right)_{i_1 i_2 \cdots i_N} = a^{(1)}_{i_1} a^{(2)}_{i_2} \cdots a^{(N)}_{i_N} \text{ for } 1 \le i_n \le I_n, n \in \mathcal{N}.$$

Sometimes the notation $\otimes$ is used (see, e.g., [23]).

2.3 Matricization of a tensor

Matricization is the rearrangement of the elements of a tensor into a matrix. Let $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ be an order-N tensor. The modes $\mathcal{N} = \{1, \ldots, N\}$ are partitioned into $\mathcal{R} = \{r_1, \ldots, r_L\}$, the modes that are mapped to the rows, and $\mathcal{C} = \{c_1, \ldots, c_M\}$, the remaining modes that are mapped to the columns. Recall that $I_{\mathcal{N}}$ denotes the set $\{I_1, \ldots, I_N\}$. Then the matricized tensor is specified by

$$\mathbf{X}_{(\mathcal{R}\times\mathcal{C}\,:\,I_{\mathcal{N}})} \in \mathbb{R}^{J\times K} \quad \text{with} \quad J = \prod_{n\in\mathcal{R}} I_n \quad \text{and} \quad K = \prod_{n\in\mathcal{C}} I_n.$$

Specifically, $\left(\mathbf{X}_{(\mathcal{R}\times\mathcal{C}\,:\,I_{\mathcal{N}})}\right)_{jk} = x_{i_1 i_2 \cdots i_N}$ with

$$j = 1 + \sum_{\ell=1}^{L}\left[(i_{r_\ell} - 1)\prod_{\ell'=1}^{\ell-1} I_{r_{\ell'}}\right] \quad \text{and} \quad k = 1 + \sum_{m=1}^{M}\left[(i_{c_m} - 1)\prod_{m'=1}^{m-1} I_{c_{m'}}\right].$$

Other notation is used in the literature. For example, $\mathbf{X}_{(\{1,2\}\times\{3,\ldots,N\}\,:\,I_{\mathcal{N}})}$ is more typically written as

$$\mathbf{X}^{I_1 I_2 \times I_3 I_4 \cdots I_N} \quad \text{or} \quad \mathbf{X}_{(I_1 I_2 \times I_3 I_4 \cdots I_N)}.$$

The main nuance in our notation is that we explicitly indicate the tensor dimensions, $I_{\mathcal{N}}$. This matters in some situations; see, e.g., (10).

Two special cases have their own notation. If $\mathcal{R}$ is a singleton, then the fibers of mode n are aligned as the columns of the resulting matrix; this is called the mode-n matricization or unfolding. The result is denoted by

$$\mathbf{X}_{(n)} \equiv \mathbf{X}_{(\mathcal{R}\times\mathcal{C}\,:\,I_{\mathcal{N}})} \text{ with } \mathcal{R} = \{n\} \text{ and } \mathcal{C} = \{1, \ldots, n-1, n+1, \ldots, N\}. \tag{1}$$

Different authors use different orderings for $\mathcal{C}$; see, e.g., [11] versus [22]. If $\mathcal{R} = \mathcal{N}$, the result is a vector and is denoted by

$$\operatorname{vec}(\mathcal{X}) \equiv \mathbf{X}_{(\mathcal{N}\times\emptyset\,:\,I_{\mathcal{N}})}. \tag{2}$$

Just as there is row and column rank for matrices, it is possible to define the mode-n rank for a tensor [11]. The n-rank of a tensor $\mathcal{X}$ is defined as

$$\operatorname{rank}_n(\mathcal{X}) = \operatorname{rank}\left(\mathbf{X}_{(n)}\right).$$

This is not to be confused with the notion of tensor rank, which is defined in §2.6.

2.4 Norm and inner product of a tensor

The inner (or scalar) product of two tensors $\mathcal{X}, \mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is defined as

$$\langle \mathcal{X}, \mathcal{Y} \rangle \equiv \operatorname{vec}(\mathcal{X})^{\mathsf{T}} \operatorname{vec}(\mathcal{Y}) = \sum_{i_1=1}^{I_1} \sum_{i_2=1}^{I_2} \cdots \sum_{i_N=1}^{I_N} x_{i_1 i_2 \cdots i_N} y_{i_1 i_2 \cdots i_N},$$

and the Frobenius norm is defined as usual: $\| \mathcal{X} \|^2 = \langle \mathcal{X}, \mathcal{X} \rangle$.

2.5 Tensor multiplication

The n-mode matrix product [11] defines multiplication of a tensor with a matrix in mode n. Let $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and $\mathbf{A} \in \mathbb{R}^{J \times I_n}$. Then

$$\mathcal{Y} = \mathcal{X} \times_n \mathbf{A} \quad \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N}$$

is defined most easily in terms of the mode-n unfolding:

$$\mathbf{Y}_{(n)} = \mathbf{A}\mathbf{X}_{(n)}. \tag{3}$$

The n-mode vector product defines multiplication of a tensor with a vector in mode n. Let $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and $\mathbf{a} \in \mathbb{R}^{I_n}$. Then

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

is tensor of order $(N-1)$, defined elementwise as

$$(\mathcal{Y})_{i_1 \cdots i_{n-1} i_{n+1} \cdots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 i_2 \cdots i_N} \, a_{i_n}.$$

More general concepts of tensor multiplication can be defined; see [4].

2.6 Tensor decompositions

As mentioned in the introduction, there are two standard tensor decompositions that are considered in this paper. Let $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$. The Tucker decomposition [49] approximates $\mathcal{X}$ as

$$\mathcal{X} \approx \mathcal{G} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \cdots \times_N \mathbf{U}^{(N)}, \tag{4}$$

where $\mathcal{G} \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times J_N}$ and $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times J_n}$ for all $n = 1, \ldots, N$. If $J_n = \operatorname{rank}_n(\mathcal{X})$ for all n, then the approximation is exact and the computation is trivial. More typically, an alternating least squares (ALS) approach is used for the computation; see [26, 45, 12]. The Tucker decomposition is not unique, but measures can be taken to correct this [19, 20, 21, 46]. Observe that the right-hand-side of (4) is a Tucker tensor, to be discussed in more detail in §4.

The CANDECOMP/PARAFAC decomposition was simultaneously developed as the canonical decomposition of Carroll and Chang [8] and the parallel factors model of Harshman [18]; it is henceforth referred to as CP per Kiers [22]. It approximates the tensor $\mathcal{X}$ as

$$\mathcal{X} \approx \sum_{r=1}^{R} \lambda_r \, \mathbf{v}_r^{(1)} \circ \mathbf{v}_r^{(2)} \circ \cdots \circ \mathbf{v}_r^{(N)}, \tag{5}$$

for some integer $R > 0$ with, for $r = 1, \ldots, R$, $\lambda_r \in \mathbb{R}$ and $\mathbf{v}_r^{(n)} \in \mathbb{R}^{I_n}$ for $n = 1, \ldots, N$. The scalar multiplier λ_r is optional and can be absorbed into one of the factors, e.g., $\mathbf{v}_1^{(r)}$. The rank of $\mathcal{X}$ is defined as the minimal R such that $\mathcal{X}$ can be exactly reproduced [27]. The right-hand side of (5) is a Kruskal tensor, which is discussed in more detail in §5.

The CP decomposition is also computed via an ALS algorithm; see, e.g., [42, 48]. Here we briefly discuss a critical part of the CP-ALS computation that can and should be specialized to sparse and factored tensors. Without loss of generality, we assume $\lambda_r = 1$ for all $r = 1, \ldots, R$. The CP model can be expressed in matrix form as

$$\mathbf{X}_{(n)} = \mathbf{V}^{(n)} \underbrace{\left(\mathbf{V}^{(N)} \odot \cdots \odot \mathbf{V}^{(n+1)} \odot \mathbf{V}^{(n-1)} \odot \cdots \odot \mathbf{V}^{(1)}\right)^{\mathsf{T}}}_{\mathbf{W}},$$

where $\mathbf{V}^{(n)} = \begin{bmatrix} \mathbf{v}_1^{(n)} & \cdots & \mathbf{v}_R^{(n)} \end{bmatrix}$ for $n = 1, \ldots, N$. If we fix everything by $\mathbf{V}^{(n)}$, then solving for it is a linear least squares problem. The pseudoinverse of the Khatri-Rao product $\mathbf{W}$ has special structure [6, 47]:

$$\begin{aligned} \mathbf{W}^{\dagger} &= \left(\mathbf{V}^{(N)} \odot \cdots \odot \mathbf{V}^{(n+1)} \odot \mathbf{V}^{(n-1)} \odot \cdots \odot \mathbf{V}^{(1)}\right) \mathbf{Z}^{\dagger} \quad \text{where} \\ \mathbf{Z} &= \left(\mathbf{V}^{(1)\mathsf{T}}\mathbf{V}^{(1)}\right) * \ldots * \left(\mathbf{V}^{(n-1)\mathsf{T}}\mathbf{V}^{(n-1)}\right) * \left(\mathbf{V}^{(n+1)\mathsf{T}}\mathbf{V}^{(n+1)}\right) * \ldots * \left(\mathbf{V}^{(N)\mathsf{T}}\mathbf{V}^{(N)}\right). \end{aligned}$$

The least-squares solution is given by $\mathbf{V}^{(n)} = \mathbf{Y}\mathbf{Z}^{\dagger}$ where $\mathbf{Y} \in \mathbb{R}^{I_n \times R}$ is defined as

$$\mathbf{Y} = \mathbf{X}_{(n)} \left(\mathbf{V}^{(N)} \odot \cdots \odot \mathbf{V}^{(n+1)} \odot \mathbf{V}^{(n-1)} \odot \cdots \odot \mathbf{V}^{(1)}\right). \tag{6}$$

For CP-ALS on large-scale tensors, the calculation of $\mathbf{Y}$ is an expensive operation and needs to be specialized. We refer to (6) as "matricized-tensor-times-Khatri-Rao-product," or "mttkrp" for short.

2.7 MATLAB details

Here we briefly describe the MATLAB code for the functions discussed in this section. The Kronecker and Hadamard matrix products are called by `kron(A,B)` and `A.*B`, respectively. The Khatri-Rao product is provided by the Tensor Toolbox and called by `khatrirao(A,B)`.

Higher-order outer products are not directly supported in MATLAB but can be implemented. For instance, $\mathcal{X} = \mathbf{a} \circ \mathbf{b} \circ \mathbf{c}$, can be computed with standard functions via

```
X = reshape(kron(kron(c,b),a),I,J,K)
```

where I, J, and K are the lengths of the vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, respectively. Using the Tensor Toolbox and the properties of the Kruskal tensor, this can be done via

```
X = full(ktensor(a,b,c)).
```

Tensor n-mode multiplication is implemented in the Tensor Toolbox via the `ttm` and `ttv` commands for matrices and vectors, respectively. Implementations for dense tensors were available in the previous version of the toolbox as discussed in [4]. We describe implementations for sparse and factored forms in this paper.

Matricization of a tensor is accomplished by permuting and reshaping the elements of the tensor. Consider the example below.

```
X = rand(5,6,4,2); R = [2 3]; C = [4 1];
I = size(X); J = prod(I(R)); K = prod(I(C));
Y = reshape(permute(X,[R C]),J,K);          % convert X to matrix Y
Z = ipermute(reshape(Y,[I(R) I(C)]),[R C]); % convert back to tensor
```

In the Tensor Toolbox, this functionality is supported transparently via the `tenmat` class, which is a generalization of a MATLAB matrix. The class stores additional information to support conversion back to a `tensor` object as well as to support multiplication with another `tenmat` object for subsequent conversion back into a `tensor` object. These features are fundamental to supporting tensor multiplication. Suppose that a tensor $\mathcal{X}$ is stored as a `tensor` object. To compute $\mathbf{A} = \mathbf{X}_{(\mathcal{R}\times\mathcal{C}:I_N)}$, use `A = tenmat(X,R,C)`; to compute $\mathbf{A} = \mathbf{X}_{(n)}$, use `A = tenmat(X,n)`; and to compute $\mathbf{A} = \text{vec}(\mathcal{X})$, use `A = tenmat(X,[1:N])` where N is the number of dimensions of the tensor $\mathcal{X}$. This functionality is implemented in the previous version of the toolbox under the name `tensor_as_matrix` and is described in detail in [4]. Support for sparse matricization is handled with `sptenmat`, which is described in §3.3.

In the Tensor Toolbox, the inner product and norm functions are called via `innerprod(X,Y)` and `norm(X)`. Efficient implementations for the sparse and factored versions are discussed in the sections that follow.

The "matricized tensor times Khatri-Rao product" in (6) is computed via `mttkrp(X, {V1,...,VN}, n)` where n is a scalar that indicates in which mode to matricize $\mathcal{X}$ and which matrix to skip, i.e., $\mathbf{V}^{(n)}$. If $\mathcal{X}$ is dense, the tensor is matricized, the Khatri-Rao product is formed explicitly, and the two are multiplied together. Efficient implementations for the sparse and factored versions are discussed in the sections that follow.

This page intentionally left blank.

3 Sparse Tensors

A sparse tensor is tensor where most of the elements are zero; in other words, it is a tensor where efficiency in storage and computation can be realized by storing and working with only the nonzeros. We consider storage in §3.1, operations in §3.2, and MATLAB details in §3.3.

3.1 Sparse tensor storage

We consider the question of how to efficiently store sparse tensors. As background, we review the closely related topic of sparse matrix storage in §3.1.1. We then consider two paradigms for storing a tensor: compressed storage in §3.1.2 and coordinate storage in §3.1.3.

3.1.1 Review of sparse matrix storage

Sparse matrices frequently arise in scientific computing, and numerous data structures have been studied for memory and computational efficiency, in serial and parallel. See [37] for an early survey of sparse matrix indexing schemes; a contemporary reference is [40, §3.4]. Here, we focus on two storage formats that can extend to higher dimensions.

The simplest storage format is coordinate format, which stores each nonzero along with its row and column index in three separate one-dimensional arrays, which Duff and Reid [13] called "parallel arrays." For a matrix $\mathbf{A}$ of size $I \times J$ with $\mathrm{nnz}(\mathbf{A})$ nonzeros, the total storage is $3 \cdot \mathrm{nnz}(\mathbf{A})$ and the indices are not necessarily presorted.

More common is compressed sparse row (CSR) and compressed sparse column (CSC) format, which appear to have originated in [17]. The CSR format stores three one-dimensional arrays: an array of length $\mathrm{nnz}(\mathbf{A})$ with the nonzero values (sorted by row), an array of length $\mathrm{nnz}(\mathbf{A})$ with corresponding column indices, and an array of length $I + 1$ that stores the beginning (and end) of each row in the other two arrays. The total storage for CSR is $2 \cdot \mathrm{nnz}(\mathbf{A}) + I + 1$. The CSC format, also known as Harwell-Boeing format, is analogous except that rows and columns are swapped; this is the format used by MATLAB [15].[2] The CSR/CSC formats are often cited for their storage efficiency, but our opinion is that the minor reduction of storage is of secondary importance. The main advantage of CSR/CSC format is that the nonzeros are necessarily grouped by row/column, which means that operations that focus on rows/columns are more efficient while other operations become more expensive, such as element insertion and matrix transpose.

[2]Search on "sparse matrix storage" in MATLAB Help or at the website `www.mathworks.com`.

3.1.2 Compressed sparse tensor storage

Numerous higher-order analogues of CSR and CSC exist for tensors. Just as in the matrix case, the idea is that the indices are somehow sorted by a particular mode (or modes).

For a third-order tensor $\mathcal{X}$ of size $I \times J \times K$, one straightforward idea is to store each frontal slice, $\mathbf{X}_{::k}$, as a sparse matrix in, say, CSC format. The entries are consequently sorted first by the third index and then by the second index.

Another idea, proposed by Lin et al. [33, 32], is to use extended Karnaugh map representation (EKMR). In this case, a three- or four-dimensional tensor is converted to a matrix (see §2.3) and then stored using a standard sparse matrix scheme, such as CSR or CSC. For example, if $\mathcal{X}$ is a three-way tensor of size $I \times J \times K$, then the EKMR scheme stores $\mathbf{X}_{(\{1\}\times\{2,3\})}$, which is a sparse matrix of size $I \times JK$. EKMR stores a fourth-order tensor as $\mathbf{X}_{(\{1,4\}\times\{2,3\})}$. Higher-order tensors are stored as a one-dimensional array (which encodes indices from the leading $n-4$ dimensions using a Karnaugh map) pointing to $n-4$ sparse four-dimensional tensors.

Lin et al. [32] compare the EKMR scheme to the method described above, i.e., storing two-dimensional slices of the tensor in CSR or CSC format. They consider two operations for the comparison: tensor addition and slice multiplication. The latter operation is multiplying subtensors (matrices) of two tensors $\mathcal{A}$ and $\mathcal{B}$, such that $\mathbf{C}_{::k} = \mathbf{A}_{::k}\mathbf{B}_{::k}$, which is matrix-matrix multiplication on the horizontal slices. In this comparison, the EKMR scheme is more efficient.

Despite these promising results, our opinion is that compressed storage is, in general, not the best option for storing sparse tensors. First, consider the problem of choosing the sort order for the indices, which is really what a compressed format boils down to. For matrices, there are only two cases: rowwise or columnwise. For an N-way tensor, however, there are $N!$ possible orderings on the modes. Second, the code complexity grows with the number of dimensions. It is well known that CSC/CSR formats require special code to handle rowwise and columnwise operations; for example, two distinct codes are needed to calculate $\mathbf{Ax}$ and $\mathbf{A}^{\mathsf{T}}\mathbf{x}$. The analogue for an Nth-order tensor would be a different code for $\mathbf{A} \;\bar{\times}_n\; \mathbf{n}$ for $n = 1, \ldots, N$. General tensor-tensor multiplication (see [4] for details) would be hard to handle. Third, we face the potential of integer overflow if we compress a tensor in a way that leads to one dimension being too big. For example, in MATLAB, indices are signed 32-bit integers, and so the largest such number is $2^{31}-1$. Storing a tensor $\mathcal{X}$ of size $2048 \times 2048 \times 2048 \times 2048$ as the (unfolded) sparse matrix $\mathbf{X}_{(1)}$ means that the number of columns is 2^{33} and consequently too large to be indexed within MATLAB. Finally, as a general rule, the idea that the data is sorted by a particular mode becomes less and less useful as the number of modes increases. Consequently, we opt for coördinate storage format, discussed in more detail below.

Before moving on, we note that there are many cases where specialized storage formats such as EKMR can be quite useful. In particular, if the number of tensor modes is relatively small (3rd- or 4th-order) and the operations are specific, e.g., only operations on frontal slices, then formats such as EKMR are likely a good choice.

3.1.3 Coordinate sparse tensor storage

As mentioned previously, we focus on coordinate storage in this paper. For a sparse tensor $\mathcal{X}$ of size $I_1 \times I_2 \times \cdots \times I_N$ with $\mathrm{nnz}(\mathcal{X})$ nonzeros, this means storing each nonzero along with its corresponding index. The nonzeros are stored in a real array of length $\mathrm{nnz}(\mathcal{X})$, and the indices are stored in an integer matrix with $\mathrm{nnz}(TX)$ rows and N columns (one per mode). The total storage is $(N+1) \cdot \mathrm{nnz}(\mathcal{X})$. We make no assumption on how the nonzeros are sorted. To the contrary, in §3.2, we show that for certain operations we can entirely avoid sorting the nonzeros.

The advantage of coordinate format is its simplicity and flexibility. Operations such as insertion are O(1). Moreover, the operations are independent of how the nonzeros are sorted, meaning that the functions need not be specialized for different mode orderings.

3.2 Operations on sparse tensors

As motivated in the previous section, we consider only the case of a sparse tensor stored in coordinate format. We consider a sparse tensor

$$\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N} \quad \text{stored as} \quad \mathbf{v} \in \mathbb{R}^P \text{ and } \mathbf{S} \in \mathbb{R}^{P \times N} \tag{7}$$

where $P = \mathrm{nnz}(\mathcal{X})$, $\mathbf{v}$ is a vector storing the nonzero values of $\mathcal{X}$, and $\mathbf{S}$ stores the subscripts corresponding to the pth nonzero as its pth row. For convenience, the subscript of the pth nonzero in dimension n is denoted by s_{p_n}. In other words, the pth nonzero is

$$x_{s_{p1}, s_{p2}, \ldots, s_{pN}} = v_p.$$

Duplicate subscripts are not allowed.

3.2.1 Assembling a sparse tensor

To assemble a sparse tensor, we require a list of nonzero values and the corresponding subscripts as input. Here, we consider the issue of resolving duplicate subscripts in that list. Typically, we simply sum the values at duplicate subscripts; for example

$$\begin{array}{ll} (2,3,4,5) & 3.4 \\ (2,3,5,5) & 4.7 \\ (2,3,4,5) & 1.1 \end{array} \quad \rightarrow \quad \begin{array}{ll} (2,3,4,5) & 4.5 \\ (2,3,5,5) & 4.7 \end{array}$$

If any subscript resolves to a value of zero, then that value and its corresponding subscript are removed.

Summation is not the only option for handling duplicate subscripts on input. We can use any rule to combine a list of values associated with a single subscript, such as max, mean, standard deviation, or even the ordinal count, as shown here:

$$\begin{array}{ll} (2,3,4,5) & 3.4 \\ (2,3,5,5) & 4.7 \\ (2,3,4,5) & 1.1 \end{array} \quad \longrightarrow \quad \begin{array}{ll} (2,3,4,5) & 2 \\ (2,3,5,5) & 1 \end{array}$$

Overall, the work of assembling a tensor reduces to finding all the unique subscripts and applying a reduction function (to resolve duplicate subscripts). The amount of work for this computation depends on the implementation, but is no worse than the cost of sorting all the subscripts, i.e., $O(P \log P)$ where $P = \text{nnz}(\mathcal{X})$.

3.2.2 Arithmetic on sparse tensors

Consider two same-sized sparse tensors $\mathcal{X}$ and $\mathcal{Y}$, stored as $(\mathbf{v}_{\mathcal{X}}, \mathbf{S}_{\mathcal{X}})$ and $(\mathbf{v}_{\mathcal{Y}}, \mathbf{S}_{\mathcal{Y}})$ as defined in (7). To compute $\mathcal{Z} = \mathcal{X} + \mathcal{Y}$, we create

$$\mathbf{v}_{\mathcal{Z}} = \begin{bmatrix} \mathbf{v}_{\mathcal{X}} \\ \mathbf{v}_{\mathcal{Y}} \end{bmatrix}, \quad \text{and} \quad \mathbf{S}_{\mathcal{Z}} = \begin{bmatrix} \mathbf{S}_{\mathcal{X}} \\ \mathbf{S}_{\mathcal{Y}} \end{bmatrix}.$$

To produce $\mathcal{Z}$, the nonzero values, $\mathbf{v}_{\mathcal{Z}}$, and corresponding subscripts, $\mathbf{S}_{\mathcal{Z}}$, are assembled by summing duplicates (see §3.2.1). Clearly, $\text{nnz}(\mathcal{Z}) \leq \text{nnz}(\mathcal{X}) + \text{nnz}(\mathcal{Y})$. In fact, $\text{nnz}(\mathcal{Z}) = 0$ if $\mathcal{Y} = -\mathcal{X}$.

It is possible to perform logical operations on sparse tensors in a similar fashion. For example, computing $\mathcal{Z} = \mathcal{X} \wedge \mathcal{Y}$ ("logical and") reduces to finding the intersection of the nonzero indices for $\mathcal{X}$ and $\mathcal{Y}$. In this case, the reduction formula is that the final value is 1 (true) only if the number of elements is at least two; for example,

$$\begin{array}{ll} (2,3,4,5) & 3.4 \\ (2,3,5,5) & 4.7 \\ (2,3,4,5) & 1.1 \end{array} \quad \longrightarrow \quad (2,3,4,5) \;\; 1 \text{ (true)}$$

For "logical and", $\text{nnz}(\mathcal{Z}) \leq \text{nnz}(\mathcal{X}) + \text{nnz}(\mathcal{Y})$. Some logical operations, however, do not produce sparse results. For example, $\mathcal{Z} = \neg\mathcal{X}$ ("logical not") has nonzeros everywhere that $\mathcal{X}$ has a zero.

Comparisons can also produce dense or sparse results. For instance, if $\mathcal{X}$ and $\mathcal{Y}$ have the same sparsity pattern, then $\mathcal{Z} = (\mathcal{X} < \mathcal{Y})$ is such that $\text{nnz}(\mathcal{Z}) \leq \text{nnz}(\mathcal{X})$. Comparison against a scalar can produce a dense or sparse result. For example, $\mathcal{Z} = (\mathcal{X} > 1)$ has no more nonzeros than $\mathcal{X}$, whereas $\mathcal{Z} = (\mathcal{X} > -1)$ has nonzeros everywhere that $\mathcal{X}$ has a zero.

3.2.3 Norm and inner product for a sparse tensor

Consider a sparse tensor $\mathcal{X}$ as in (7) with $P = \text{nnz}(\mathcal{X})$. The work to compute the norm is $O(P)$ and does not involve any data movement:

$$\| \mathcal{X} \| = \sqrt{\sum_{p=1}^{P} v_p^2} \; .$$

The inner product of two same-sized sparse tensors, $\mathcal{X}$ and $\mathcal{Y}$, involves finding duplicates in their subscripts, similar to the problem of assembly (see §3.2.1). The cost is no worse than the cost of sorting all the subscripts, i.e., $O(P \log P)$ where $P = \text{nnz}(\mathcal{X}) + \text{nnz}(\mathcal{Y})$.

3.2.4 n-mode vector multiplication for a sparse tensor

Coordinate storage format is amenable to the computation of a tensor times a vector in mode n. We can do this computation in $O(\text{nnz}(\mathcal{X}))$ time, though this does not account for the cost of data movement, which is generally the most time-consuming part of this operation. (The same is true for sparse matrix-vector multiplication.)

Consider

$$\mathcal{Y} = \mathcal{X} \,\bar{\times}_n\, \mathbf{a},$$

where $\mathcal{X}$ is as defined in (7) and the vector $\mathbf{a}$ is of length I_n, For each $p = 1, \ldots, P$, nonzero v_p is multiplied by $a_{s_{pn}}$ and added to the $(s_{p1}, \ldots, s_{pn-1}, s_{pn+1}, \ldots, s_{pN})$ element of $\mathcal{Y}$. Stated another way, we can convert $\mathbf{a}$ to an "expanded" vector $\mathbf{b} \in \mathbb{R}^P$ such that

$$b_p = a_{s_{np}} \text{ for } p = 1, \ldots, P.$$

Next we can calculate a vector of values $\hat{\mathbf{v}} \in \mathbb{R}^P$ so that

$$\hat{\mathbf{v}} = \mathbf{v} * \mathbf{b}.$$

We create a matrix $\hat{\mathbf{S}}$ that is equal to $\mathbf{S}$ with the nth column removed. Then the nonzeros $\hat{\mathbf{v}}$ and subscripts $\hat{\mathbf{S}}$ can be assembled (summing duplicates) to create $\mathcal{Y}$. Observe that $\text{nnz}(\mathcal{Y}) \leq \text{nnz}(\mathcal{X})$, but the number of dimensions has also reduced by one, meaning the the final result is not necessarily sparse even though the number of nonzeros cannot increase.

We can generalize the previous discussion to multiplication by vectors in multiple modes. For example, consider the case of multiplication in every mode:

$$\alpha = \mathcal{X} \,\bar{\times}_1\, \mathbf{a}^{(1)} \cdots \,\bar{\times}_N\, \mathbf{a}^{(N)}.$$

Define "expanded" vectors $\mathbf{b}^{(n)} \in \mathbb{R}^P$ for $n = 1, \ldots, N$ such that

$$b_p^{(n)} = a_{s_{np}}^{(n)} \text{ for } p = 1, \ldots, P.$$

We then calculate $\mathbf{w} = \mathbf{v} * \mathbf{b}^{(1)} * \cdots * \mathbf{b}^{(N)}$, and the final scalar result is $\alpha = \sum_{p=1}^{P} w_p$. Observe that we calculate all the n-mode products simultaneously rather than in sequence. Hence, only one "assembly" of the final result is needed.

3.2.5 n-mode matrix multiplication for a sparse tensor

The computation of a sparse tensor times a matrix in mode n is straightforward. To compute

$$\mathcal{Y} = \mathcal{X} \times_n \mathbf{A},$$

we use the matricized version in (3), storing $\mathbf{X}_{(n)}$ as a sparse matrix. As one might imagine, CSR format works well for mode-n unfoldings, but CSC format does not because there are so many columns. For CSC, use the transposed version of the equation, i.e.,

$$\mathbf{Y}_{(n)}^{\mathsf{T}} = \mathbf{X}_{(n)}^{\mathsf{T}} \mathbf{A}^{\mathsf{T}}.$$

Unless $\mathbf{A}$ has special structure (e.g., diagonal), the result is dense. Consequently, this only works for relatively small tensors (and is why we have glossed over the possibility of integer overflow when we convert $\mathcal{X}$ to $\mathbf{X}_{(n)}$). The cost boils down to that of converting $\mathcal{X}$ to a sparse matrix, doing a matrix-by-sparse-matrix multiply, and converting the result into a (dense) tensor $\mathcal{Y}$. Multiple n-mode matrix multiplications are performed sequentially.

3.2.6 General tensor multiplication for sparse tensors

For tensor-tensor multiplication, the modes to be multiplied are specified. For example, if we have two tensors $\mathcal{X} \in \mathbb{R}^{3\times4\times5}$ and $\mathcal{Y} \in \mathbb{R}^{4\times3\times2\times2}$, we can calculate:

$$\mathcal{Z} = \langle \mathcal{X}, \mathcal{Y} \rangle_{\{1,2;2,1\}} \in \mathbb{R}^{5\times2\times2}$$

which means that we multiply modes 1 and 2 of $\mathcal{X}$ with modes 2 and 1 of $\mathcal{Y}$. Here, we refer to the modes that are being multiplied as the "inner" modes and the other modes as the "outer" modes because, in essence, we are taking inner and outer products along these modes. Because it takes several pages to explain tensor-tensor multiplication, we have omitted it from the background material in §2 and instead refer the interested reader to [4].

In the sparse case, we have to find all the matches of the inner modes of $\mathcal{X}$ and $\mathcal{Y}$, compute the Kronecker product of the matches, associate each element of the product with a subscript that comes from the outer modes, and then resolve duplicate subscripts by summing the corresponding nonzeros. Depending on the modes specified, the work can be as high as $O(PQ)$, where $P = \text{nnz}(\mathcal{X})$ and $Q = \text{nnz}(\mathcal{Y})$, but can be closer to $O(P \log P + Q \log Q)$ depending on which modes are multiplied and the structure on the nonzeros.

3.2.7 Matricized sparse tensor times Khatri-Rao product

Consider the calculation of the matricized tensor times a Khatri-Rao product in (6). We compute this indirectly using the n-mode vector multiplication, which is efficient for large, sparse tensors (see §3.2.4), by rewriting (6) as

$$\mathbf{w}_r = \mathcal{X} \,\bar{\times}_1\, \mathbf{v}_r^{(1)} \cdots \,\bar{\times}_{n-1}\, \mathbf{v}_r^{(n-1)} \,\bar{\times}_{n+1}\, \mathbf{v}_r^{(n+1)} \cdots \,\bar{\times}_N\, \mathbf{v}_r^{(N)} \quad \text{for } r = 1, 2, \ldots, R.$$

In other words, the solution $\mathbf{W}$ is computed column-by-column. The cost equates to computing the product of the sparse tensor with $N-1$ vectors R times.

3.2.8 Computing $\mathbf{X}_{(n)}\mathbf{X}_{(n)}^{\mathsf{T}}$ for a sparse tensor

Generally, the product $\mathbf{Z} = \mathbf{X}_{(n)}\mathbf{X}_{(n)}^{\mathsf{T}} \in \mathbb{R}^{I_n \times I_n}$ can be computed directly by storing $\mathbf{X}_{(n)}$ as a sparse matrix. As in §3.2.5, we must be wary of CSC format, in which case we should actually store $\mathbf{A} = \mathbf{X}_{(n)}^{\mathsf{T}}$ and then calculate $\mathbf{Z} = \mathbf{A}^{\mathsf{T}}\mathbf{A}$. The cost is primarily the cost of converting to a sparse matrix format (e.g., CSC) plus the matrix-matrix multiply to form the dense matrix $\mathbf{Z} \in \mathbb{R}^{I_n \times I_n}$. However, the matrix $\mathbf{X}_{(n)}$ is of size

$$I_n \times \prod_{\substack{m=1\\ m\neq n}}^{N} I_m,$$

which means that its column indices may overflow the integers is the tensor dimensions are very big.

3.2.9 Collapsing and scaling on sparse tensors

We present the concepts of collapsing and scaling on tensors to extend well-known (and mostly unnamed) operations on matrices.

For a matrix, one might want to compute the sum of all elements in each row, or the maximum element in each column, or the average of all elements, and so on. To the best of our knowledge, these sorts of operations do not have a name, so we call them *collapse* operations—we are collapsing the object in one or more dimensions to get some statistical information. Conversely, we often want to use the results of a collapse operation to *scale* the elements of a matrix. For example, to convert a matrix $\mathbf{A}$ to a row-stochastic matrix, we compute the collapsed sum in mode 1 (rowwise) and call it $\mathbf{z}$, and then scale $\mathbf{A}$ in mode 1 by $(1/\mathbf{z})$.

We can define similar operations in the N-way context for tensors. For collapsing, we define the modes to be collapsed and the operation (e.g., sum, max, number of elements, etc.). Likewise, scaling can be accomplished by specifying the modes to scale.

Suppose, for example, that we have an $I \times J \times K$ tensor $\mathcal{X}$ and want to scale each frontal slice so that its largest entry is one. First, we collapse the tensor in modes 1 and 2 using the max operation. In other words, we compute the maximum of each frontal slice, i.e.,

$$z_k = \max\{x_{ijk} \mid i = 1, \ldots, I \text{ and } j = 1, \ldots, J\} \quad \text{for } k = 1, \ldots, K.$$

This is accomplished in coordinate format by considering only the third subscript corresponding to each nonzero, doing assembly with duplicate resolution via the appropriate collapse operation (in this case, max). Then the scaled tensor can be computed elementwise by

$$y_{ijk} = \frac{x_{ijk}}{z_k}.$$

This computation can be completed by "expanding" $\mathbf{z}$ to a vector of length nnz(X) as was done for the sparse-tensor-times-vector operation in §3.2.4.

3.3 MATLAB details for sparse tensors

MATLAB does not natively support sparse tensors. In the Tensor Toolbox, sparse tensors are stored in the `sptensor` class, which stores the size as an integer N-vector along with the vector of nonzero values $\mathbf{v}$ and corresponding integer matrix of subscripts $\mathbf{S}$ from (7).

We can assemble a sparse tensor from a list of subscripts and corresponding values, as described in §3.2.1. By default, we sum repeated entries, though we allow the option of using other functions to resolve duplicates. To this end, we rely on the MATLAB `accumarray` function, which takes a list of subscripts, a corresponding list of values, and a function to resolve the duplicates (sum, be default). To use this with large-scale sparse data is complex. We first calculate a codebook of the Q unique subscripts (using the MATLAB `unique` function), use the codebook to convert each N-way subscript to an integer value between 1 and Q, call `accumarray` with the integer indices, and then use the codebook to map the final result back to the corresponding N-way subscripts.

MATLAB relies heavily on *linear indices* for any operation that returns a list of subscripts. For example, the `find` command on a sparse matrix returns linear indices (by default) that can be subsequently be converted to row and column indices. For tensors, we are wary of linear indices due to the possibility of integer overflow discussed in §3.1.2. Specifically, linear indices may produce integer interflow if the product of the dimensions of the tensor is greater than or equal to 2^{32}, e.g., a four-way tensor of size $2048 \times 2048 \times 2048 \times 2048$. Thus, our versions of subscripted reference (`subsref`) and assignment (`subsasgn`) as well as our version of `find` explicitly use subscripts and do not support linear indices.

We do, however, support the conversion of a sparse tensor to a matrix stored in coordinate format via the class sptenmat. This matrix can then be converted into a MATLAB sparse matrix via the command double.

All operations are called in the same way for sparse tensors as they are for dense tensor, e.g., Z = X + Y. Logical operations always produce sptensor results, even if they would be more efficiently stored as dense tensors. To convert to a dense tensor, call full(X).

The three multiplication operations may produce dense results: tensor-times-tensor (ttt), tensor-times-matrix (ttm) and tensor-times-vector (ttv). In the case of ttm, since it is called repeatedly for multiplication in multiple modes, any intermediate product may be dense and the remaining calls will be to the dense version of ttm. For general tensor multiplication, which reduces to sparse matrix-matrix multiplication, we take measures to avoid integer overflow by instead finding the unique subscripts and only using that many rows/columns in the matrices that are multiplied. This is similar to how we use accumarray to assemble a tensor.

Generating a random sparse tensor is complicated because it requires generating the locations of the nonzeros as well as the nonzeros. Thus, the Tensor Toolbox provides the command sptenrand(sz,nnz) to produce a sparse tensor. It is analogous to the command sprand to produce a random sparse matrix in MATLAB with two exceptions. First, the size is passed in as a single (row vector) input. Second, the last argument can be either a percentage (as in sprand) or an explicit number of nonzeros desired. We also provide a function sptendiag to create a superdiagonal tensor.

This page intentionally left blank.

4 Tucker Tensors

Consider a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ such that

$$\mathcal{X} = [\![\mathcal{G} \, ; \mathbf{U}^{(1)}, \mathbf{U}^{(2)}, \ldots, \mathbf{U}^{(N)}]\!] \equiv \mathcal{G} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \cdots \times_N \mathbf{U}^{(N)}, \tag{8}$$

where $\mathcal{G} \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times J_N}$ is the core tensor and $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times J_n}$ for $n = 1, \ldots, N$. This is the format that results from a Tucker decomposition [49] and is therefore termed a *Tucker tensor.* We use the shorthand notation $[\![\mathcal{G} \, ; \mathbf{U}^{(1)}, \mathbf{U}^{(2)}, \ldots, \mathbf{U}^{(N)}]\!]$ from [24], but other notation can be used. For example, Lim [31] proposes that the covariant aspect of the multiplication be made explicit by expressing (8) as

$$\left(\mathbf{U}^{(1)}, \mathbf{U}^{(2)}, \ldots, \mathbf{U}^{(N)} \right) \cdot \mathcal{G}.$$

As another example, Grigorascu and Regalia [16] emphasize the role of the core tensor in the multiplication by expressing (8) as

$$\mathcal{X} = \mathbf{U}^{(1)} \overset{\mathcal{G}}{\star} \mathbf{U}^{(2)} \overset{\mathcal{G}}{\star} \cdots \overset{\mathcal{G}}{\star} \mathbf{U}^{(N)},$$

which is called the *weighted Tucker product*; the unweighted version has $\mathcal{G} = \mathcal{I}$, the identity tensor. Regardless of the notation, the properties of a Tucker tensor are the same.

4.1 Tucker tensor storage

Storing $\mathcal{X}$ as a Tucker tensor can have major advantages in terms of memory requirements. In its explicit form, $\mathcal{X}$ requires storage of

$$\prod_{n=1}^{N} I_n \qquad \text{versus} \qquad \textsc{storage}(\mathcal{G}) + \sum_{n=1}^{N} I_n J_n$$

elements for the factored form. Thus, the Tucker tensor factored format is clearly advantageous if $\textsc{storage}(\mathcal{G})$ is sufficiently small. This certainly is the case if

$$\prod_{n=1}^{N} J_n \ll \prod_{n=1}^{N} I_n. \tag{9}$$

However, there is no reason to assume that the core tensor $\mathcal{G}$ is dense; on the contrary, $\mathcal{G}$ might itself be sparse or factored. The next section discusses computations on $\mathcal{X}$ in its factored form, making minimal assumptions about the format of $\mathcal{G}$.

4.2 Tucker tensor properties

It is common knowledge (dating back to [49]) that matricized versions of the Tucker tensor (8) have a special form; specifically,

$$\mathbf{X}_{(\mathcal{R}\times\mathcal{C}\,:\,J_{\mathcal{N}})} = \left(\mathbf{U}^{(r_L)} \otimes \cdots \otimes \mathbf{U}^{(r_1)}\right) \mathbf{G}_{(\mathcal{R}\times\mathcal{C}\,:\,I_{\mathcal{N}})} \left(\mathbf{U}^{(c_M)} \otimes \cdots \otimes \mathbf{U}^{(c_1)}\right)^{\mathsf{T}}, \tag{10}$$

where $\mathcal{R} = \{r_1, \ldots, r_L\}$ and $\mathcal{C} = \{c_1, \ldots, c_M\}$. Note that the order of the indices in $\mathcal{R}$ and $\mathcal{C}$ does matter, and reversing the order of the indices is a frequent source of coding errors. For the special case of mode-n matricization (1), we have

$$\mathbf{X}_{(n)} = \mathbf{U}^{(n)}\mathbf{G}_{(n)} \left(\mathbf{U}^{(N)} \otimes \cdots \otimes \mathbf{U}^{(n+1)} \otimes \mathbf{U}^{(n-1)} \otimes \cdots \otimes \mathbf{U}^{(1)}\right)^{\mathsf{T}}. \tag{11}$$

Likewise, for the vectorized version (2), we have

$$\operatorname{vec}(\mathcal{X}) = \left(\mathbf{U}^{(N)} \otimes \cdots \otimes \mathbf{U}^{(1)}\right) \operatorname{vec}(\mathcal{G}). \tag{12}$$

4.2.1 n-mode matrix multiplication for a Tucker tensor

Multiplying a Tucker tensor times a matrix in mode n reduces to multiplying its nth factor matrix; in other words, the result retains the factored Tucker tensor structure. Let $\mathcal{X}$ be as in (8) and $\mathbf{V}$ be a matrix of size $K \times I_n$. Then from (3) and (11), we have

$$\mathcal{X} \times_n \mathbf{V} = [\![\mathcal{G}\,; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(n-1)}, \mathbf{V}\mathbf{U}^{(n)}, \mathbf{U}^{(n+1)}, \ldots, \mathbf{U}^{(N)}]\!].$$

The cost is that of the matrix-matrix multiply, that is, $O(I_n J_n K)$. More generally, let $\mathbf{V}^{(n)}$ be of size $K_n \times I_n$ for $n = 1, \ldots, N$. Then

$$[\![\mathcal{X}\,; \mathbf{V}^{(1)}, \ldots, \mathbf{V}^{(N)}]\!] = [\![\mathcal{G}\,; \mathbf{V}^{(1)}\mathbf{U}^{(1)}, \ldots, \mathbf{V}^{(N)}\mathbf{U}^{(N)}]\!].$$

The cost here is the cost of N matrix-matrix multiplies, for a total of $O(\sum_n I_n J_n K_n)$, and the Tucker tensor structure is retained. As an aside, if $\mathbf{U}^{(n)}$ has full column rank and $\mathbf{V}^{(n)} = \mathbf{U}^{(n)\dagger}$ for $n = 1, \ldots, N$, then $\mathcal{G} = [\![\mathcal{X}\,; \mathbf{U}^{(1)\dagger}, \ldots, \mathbf{U}^{(N)\dagger}]\!]$.

4.2.2 n-mode vector multiplication for a Tucker tensor

Multiplication of a Tucker tensor by a vector follows similar logic to the matrix case except that the nth factor matrix necessarily disappears and the problem reduces to n-mode vector multiplication with the core. Let $\mathcal{X}$ be a Tucker tensor as in (8) and $\mathbf{v}$ be a vector of size I_n; then,

$$\mathcal{X}\ \bar{\times}_n\ \mathbf{v} = [\![\mathcal{G}\ \bar{\times}_n\ \mathbf{w}\,; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(n-1)}, \mathbf{U}^{(n+1)}, \ldots, \mathbf{U}^{(N)}]\!] \quad \text{where} \quad \mathbf{w} = \mathbf{U}^{(n)\mathsf{T}}\mathbf{v}.$$

The cost here is that of multiplying a matrix times a vector, $O(I_n J_n)$, plus the cost of multiplying the core (which could be dense, sparse, or factored) times a vector. The

Tucker tensor structure is retained, but with one less factor matrix. More generally, multiplying a Tucker tensor by a vector in every mode converts to the problem of multiplying its core by a vector in every mode. Let $\mathbf{V}^{(n)}$ be of size I_n for $n = 1, \ldots, N$; then

$$\mathcal{X} \bar{\times}_1 \mathbf{v}^{(1)} \cdots \bar{\times}_N \mathbf{v}^{(N)} = \mathcal{G} \bar{\times}_1 \mathbf{w}^{(1)} \cdots \bar{\times}_N \mathbf{w}^{(N)}$$
$$\text{where} \quad \mathbf{w}^{(n)} = \mathbf{U}^{(n)\mathsf{T}} \mathbf{v}^{(n)} \text{ for all } n = 1, \ldots, N.$$

In this case, the work is the cost of N matrix-vector multiplies, $\mathrm{O}(\sum_n I_n J_n)$, plus the cost of multiplying the core by a vector in each mode. If $\mathcal{G}$ is dense, the total cost is

$$\mathrm{O}\left(\sum_{n=1}^{N} \left(I_n J_n + \prod_{m=n}^{N} J_m \right) \right).$$

Further gains in efficiency are possible by doing the multiplies in order of largest to smallest J_n. The Tucker tensor structure is clearly not retained for all-mode vector multiplication.

4.2.3 Inner product

Let $\mathcal{X}$ be a Tucker tensor as in (8) and let $\mathcal{Y}$ be a Tucker tensor of the same size with

$$\mathcal{Y} = [\![\mathcal{H} \,; \mathbf{V}^{(1)}, \ldots, \mathbf{V}^{(N)}]\!]$$

with $\mathcal{H} \in \mathbb{R}^{K_1 \times K_2 \times \cdots \times K_N}$ and $\mathbf{V}^{(n)} \in \mathbb{R}^{I_n \times K_n}$ for $n = 1, \ldots, N$. If the cores are small in relation to the overall tensor size, we can realize computational savings as follows. Without loss of generality, assume $\mathcal{G}$ is smaller than (or at least no larger than) $\mathcal{H}$, e.g., $J_n \leq K_n$ for all n. Then

$$\begin{aligned}
\langle \mathcal{X}, \mathcal{Y} \rangle &= \langle [\![\mathcal{G} \,; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!], [\![\mathcal{H} \,; \mathbf{V}^{(1)}, \ldots, \mathbf{V}^{(N)}]\!] \rangle \\
&= \langle \mathcal{G}, [\![\mathcal{H} \,; \mathbf{V}^{(1)}, \ldots, \mathbf{V}^{(N)}]\!] \times_1 \mathbf{U}^{(1)\mathsf{T}} \cdots \times_N \mathbf{U}^{(N)\mathsf{T}} \rangle \\
&= \langle \mathcal{G}, [\![\mathcal{H} \,; \mathbf{U}^{(1)\mathsf{T}} \mathbf{V}^{(1)}, \ldots, \mathbf{U}^{(N)\mathsf{T}} \mathbf{V}^{(N)}]\!] \rangle \\
&= \langle \mathcal{G}, \mathcal{F} \rangle \text{ with } \mathcal{F} = [\![\mathcal{H} \,; \mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(N)}]\!] \text{ and } \mathbf{W}^{(n)} = \mathbf{U}^{(n)\mathsf{T}} \mathbf{V}^{(n)} \text{ for all } n.
\end{aligned}$$

Each $\mathbf{W}^{(n)}$ is of size $J_n \times K_n$ and costs $\mathrm{O}(I_n J_n K_n)$ to compute. Then, to compute $\mathcal{F}$, we do a tensor-times-matrix in all modes with the tensor $\mathcal{H}$ (the cost varies depending on the tensor type), followed by an inner product between two tensors of size $J_1 \times J_2 \times \cdots \times J_N$. If $\mathcal{G}$ and $\mathcal{H}$ are dense, then the total cost is

$$\mathrm{O}\left(\sum_{n=1}^{N} I_n J_n K_n + \sum_{n=1}^{N} \left(\prod_{p=n}^{N} K_p \prod_{q=1}^{n} J_q \right) + \prod_{n=1}^{N} J_n \right).$$

4.2.4 Norm of a Tucker tensor

For the previous discussion, it is clear that the norm can also be calculated efficiently if the core tensor is small in relation to the overall tensor, e.g., $J_n < I_n$ for all n. Let $\mathcal{X}$ be a Tucker tensor as in (8). From §4.2.3, we have

$$\begin{aligned}\|\mathcal{X}\|^2 &= \langle \mathcal{X}, \mathcal{X} \rangle = \langle \mathcal{G}, \mathcal{F} \rangle \\ &\quad \text{with } \mathcal{F} = [\![\mathcal{G}\,; \mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(N)}]\!] \text{ and } \mathbf{W}^{(n)} = \mathbf{U}^{(n)\mathsf{T}}\mathbf{U}^{(n)} \text{ for all } n. \qquad (13)\end{aligned}$$

Forming all the $\mathbf{W}^{(n)}$ matrices costs $O(\sum_n I_n J_n^2)$. To compute $\mathcal{F}$, we have to do a tensor-times-matrix in all N modes, and if $\mathcal{G}$ is dense, for example, the cost is $O(\prod_n J_n \cdot \sum_n J_n)$. Finally, we compute an inner product of two tensors of size $J_1 \times J_2 \times \cdots \times J_n$, which costs $O(\prod_n J_n)$ if both tensors are dense.

4.2.5 Matricized Tucker tensor times Khatri-Rao product

As noted in §2.6, a common operation is to calculate a particular matricized tensor times a special Khatri-Rao product (6). In the case of a Tucker tensor, we can reduce this to an equivalent operation on the core tensor. Let $\mathcal{X}$ be a Tucker tensor as in (8) and let $\mathbf{V}^{(m)}$ be a matrix of size $I_m \times R$ for all $m \neq n$. The goal is to calculate

$$\begin{aligned}\mathbf{W} &= \mathbf{X}_{(n)} \left(\mathbf{V}^{(N)} \odot \cdots \odot \mathbf{V}^{(n+1)} \odot \mathbf{V}^{(n-1)} \odot \cdots \odot \mathbf{V}^{(1)}\right) \\ &= \mathbf{U}^{(n)} \mathbf{G}_{(n)} \left(\mathbf{U}^{(N)} \otimes \cdots \otimes \mathbf{U}^{(n+1)} \otimes \mathbf{U}^{(n-1)} \otimes \cdots \otimes \mathbf{U}^{(1)}\right)^{\mathsf{T}} \\ &\qquad \left(\mathbf{V}^{(N)} \odot \cdots \odot \mathbf{V}^{(n+1)} \odot \mathbf{V}^{(n-1)} \odot \cdots \odot \mathbf{V}^{(1)}\right).\end{aligned}$$

Using the properties of the Khatri-Rao product [42] and setting $\mathbf{W}^{(m)} = \mathbf{U}^{(m)\mathsf{T}}\mathbf{V}^{(m)}$ for $m \neq n$, we have

$$\mathbf{W} = \mathbf{U}^{(n)} \underbrace{\mathbf{G}_{(n)} \left(\mathbf{W}^{(N)} \odot \cdots \odot \mathbf{W}^{(n+1)} \odot \mathbf{W}^{(n-1)} \odot \cdots \odot \mathbf{W}^{(1)}\right)}_{\text{Matricized core tensor } \mathcal{G} \text{ times Khatri-Rao product}}.$$

Thus, this requires $(N-1)$ matrix-matrix products to form the matrices $\mathbf{W}^{(m)}$ of size $J_m \times R$, each of which costs $O(I_m J_m R)$. Then we calculate the "mttkrp" with $\mathcal{G}$, and the cost is $O(R \prod_n J_n)$ if $\mathcal{G}$ is dense. The final matrix-matrix multiply costs $O(I_n J_n R)$. If $\mathcal{G}$ is dense, the total cost is

$$O\left(R\left(\sum_{n=1}^{N} I_n J_n + \prod_{n=1}^{N} J_n\right)\right).$$

4.2.6 Computing $\mathbf{X}_{(n)}\mathbf{X}_{(n)}^{\mathsf{T}}$ for a Tucker tensor

To compute $\text{rank}_n(\mathcal{X})$, we need $\mathbf{Z} = \mathbf{X}_{(n)}\mathbf{X}_{(n)}^{\mathsf{T}}$. Let $\mathcal{X}$ be a Tucker tensor as in (8); then

$$\begin{aligned}\mathbf{Z} = \mathbf{U}^{(n)}\mathbf{G}_{(n)} &\left(\mathbf{U}^{(N)} \otimes \cdots \otimes \mathbf{U}^{(n+1)} \otimes \mathbf{U}^{(n-1)} \otimes \cdots \otimes \mathbf{U}^{(1)}\right)^{\mathsf{T}} \\ &\left(\mathbf{U}^{(N)} \otimes \cdots \otimes \mathbf{U}^{(n+1)} \otimes \mathbf{U}^{(n-1)} \otimes \cdots \otimes \mathbf{U}^{(1)}\right) \mathbf{G}_{(n)}^{\mathsf{T}}\mathbf{U}^{(n)\mathsf{T}}.\end{aligned}$$

Using the properties of the Kronecker product, this reduces to

$$\mathbf{Z} = \mathbf{U}^{(n)}\mathbf{G}_{(n)} \left(\mathbf{V}^{(N)} \otimes \cdots \otimes \mathbf{V}^{(n+1)} \otimes \mathbf{V}^{(n-1)} \otimes \cdots \otimes \mathbf{V}^{(1)}\right) \mathbf{G}_{(n)}^{\mathsf{T}}\mathbf{U}^{(n)\mathsf{T}},$$

where $\mathbf{V}^{(m)} = \mathbf{U}^{(m)\mathsf{T}}\mathbf{U}^{(m)} \in \mathbb{R}^{J_m \times J_m}$ for all $m \neq n$ at a cost of $O(I_m J_m^2)$. Finally, this becomes the product of two matricized tensors and a third matrix:

$$\mathbf{Z} = \mathbf{H}_{(n)}\mathbf{G}_{(n)}^{\mathsf{T}}\mathbf{U}^{(n)\mathsf{T}} \quad \text{where} \quad \mathcal{H} = [\![\mathcal{G}\,; \mathbf{V}^{(1)}, \ldots, \mathbf{V}^{(n-1)}, \mathbf{U}^{(n)}, \mathbf{V}^{(n+1)}, \ldots, \mathbf{V}^{(N)}]\!].$$

If $\mathcal{G}$ is dense, forming $\mathcal{H}$ costs

$$O\left(\prod_{m=1}^{N} J_m \cdot \left(I_n + \sum_{\substack{m=1\\ m\neq n}}^{N} J_m + \right)\right).$$

And the final multiplication of the three matrices costs $O(I_n \prod_{m=1}^{N} J_m + I_n^2 J_n)$.

4.3 MATLAB details for Tucker tensors

A Tucker tensor $\mathcal{X}$ is constructed in MATLAB by passing in the core array $\mathcal{G}$ and factor matrices $\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}$ using `X = ttensor(G,{U1,...,UN})`. In version 1.0 of the Tensor Toolbox, this class was called `tucker_tensor` [4]. The core tensor can be any of the four classes of tensors supported by the Tensor Toolbox.

A Tucker tensor can be converted to a standard `tensor` by calling `full(X)`. Subscripted reference and assignment can only be done on the factors, not elementwise. For example, it is possible to change the $(1,1)$ element of $\mathbf{U}^{(2)}$ but not the $(1,1,1)$ element of a three-way Tucker tensor $\mathcal{X}$. Scalar multiplication is supported, i.e., `X*5`.

The n-mode product of a Tucker tensor with one or more matrices (§4.2.1) or vectors (§4.2.2) is implemented in `ttm` and `ttv`, respectively. The inner product (§4.2.3 and also §6) is called via `innerprod`, and the norm of a Tucker tensor is called via `norm`. The function `mttkrp` computes the matricized-tensor-times-Khatri-Rao-product as described in §4.2.5. The function `nvecs(X,n)` computes the leading mode-n eigenvectors for $\mathbf{X}_{(n)}\mathbf{X}_{(n)}^{\mathsf{T}}$ and relies on the efficiencies described in §4.2.6.

This page intentionally left blank.

5 Kruskal tensors

Consider a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ that can be written as a sum of R rank-1 tensors (with no assumption that R is minimal), i.e.,

$$\mathcal{X} = \sum_{r=1}^{R} \lambda_r \, \mathbf{u}_r^{(1)} \circ \cdots \circ \mathbf{u}_r^{(N)},$$

where $\boldsymbol{\lambda} = \begin{bmatrix} \lambda_1 & \cdots & \lambda_R \end{bmatrix}^{\mathsf{T}} \in \mathbb{R}^R$ and $\mathbf{U}^{(n)} = \begin{bmatrix} \mathbf{u}_1^{(n)} & \cdots & \mathbf{u}_R^{(n)} \end{bmatrix} \in \mathbb{R}^{I_n \times R}$. This is the format that results from a PARAFAC decomposition [18, 8], and we refer to it as a *Kruskal tensor* due to the work of Kruskal on tensors of this format [27, 28]. We use the shorthand notation from [24]:

$$\mathcal{X} = [\![\boldsymbol{\lambda} \, ; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]. \tag{14}$$

In some cases, the weights λ_r are not explicit and we write $\mathcal{X} = [\![\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$. Other notation can be used. For instance, Kruskal [27] uses

$$\mathcal{X} = \left(\mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)} \right).$$

5.1 Kruskal tensor storage

Storing $\mathcal{X}$ as a Kruskal tensor is efficient in terms of storage. In its explicit form, $\mathcal{X}$ requires storage of

$$\prod_{n=1}^{N} I_n \quad \text{versus} \quad R \left(1 + \sum_{n=1}^{N} I_n \right)$$

elements for the factored form. We do not assume that R is minimal.

5.2 Kruskal tensor properties

The Kruskal tensor is a special case of the Tucker tensor where the core tensor $\mathcal{G}$ is an $R \times R \times \cdots \times R$ diagonal tensor and all the factor matrices $\mathbf{U}^{(n)}$ have R columns.

It is well known that matricized versions of the Kruskal tensor (14) have a special form; namely,

$$\mathbf{X}_{(\mathcal{R} \times \mathcal{C} : I_N)} = \left(\mathbf{U}^{(r_L)} \odot \cdots \odot \mathbf{U}^{(r_1)} \right) \boldsymbol{\Lambda} \left(\mathbf{U}^{(c_M)} \odot \cdots \odot \mathbf{U}^{(c_1)} \right)^{\mathsf{T}},$$

where $\boldsymbol{\Lambda} = \mathrm{diag}(()\lambda)$. For the special case of mode-n matricization, this reduces to

$$\mathbf{X}_{(n)} = \mathbf{U}^{(n)} \boldsymbol{\Lambda} \left(\mathbf{U}^{(N)} \odot \cdots \odot \mathbf{U}^{(n+1)} \odot \mathbf{U}^{(n-1)} \odot \cdots \odot \mathbf{U}^{(1)} \right)^{\mathsf{T}}. \tag{15}$$

Finally, the vectorized version is

$$\mathrm{vec}(\mathcal{X}) = \left(\mathbf{U}^{(N)} \odot \cdots \odot \mathbf{U}^{(1)} \right) \boldsymbol{\lambda}. \tag{16}$$

5.2.1 Adding two Kruskal tensors

Because the Kruskal tensor is a sum of rank-1 tensors, adding two Kruskal tensors together can be viewed as extending that summation over both sets of terms. For instance, consider Kruskal tensors $\mathcal{X}$ and $\mathcal{Y}$ of the same size given by:

$$\mathcal{X} = [\![\boldsymbol{\lambda} \,; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!] \quad \text{and} \quad \mathcal{Y} = [\![\boldsymbol{\sigma} \,; \mathbf{V}^{(1)}, \ldots, \mathbf{V}^{(N)}]\!].$$

Adding $\mathcal{X}$ and $\mathcal{Y}$ yields

$$\mathcal{X} + \mathcal{Y} = \sum_{r=1}^{R} \lambda_r \, \mathbf{u}_r^{(1)} \circ \cdots \circ \mathbf{u}_r^{(N)} + \sum_{p=1}^{P} \sigma_p \, \mathbf{v}_p^{(1)} \circ \cdots \circ \mathbf{v}_p^{(N)},$$

or, alternatively,

$$\mathcal{X} + \mathcal{Y} = [\![\begin{bmatrix} \boldsymbol{\lambda} \\ \boldsymbol{\sigma} \end{bmatrix} \,; \begin{bmatrix} \mathbf{U}^{(1)} & \mathbf{V}^{(1)} \end{bmatrix}, \cdots, \begin{bmatrix} \mathbf{U}^{(N)} & \mathbf{V}^{(N)} \end{bmatrix}]\!].$$

The work for this is O(1).

5.2.2 Mode-n matrix multiplication for a Kruskal tensor

Let $\mathcal{X}$ be a Kruskal tensor as in (14) and $\mathbf{V}$ be a matrix of size $J \times I_n$. From the definition of mode-n matrix multiplication and (15), we have

$$\mathcal{X} \times_n \mathbf{V} = [\![\boldsymbol{\lambda} \,; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(n-1)}, \mathbf{V}\mathbf{U}^{(n)}, \mathbf{U}^{(n+1)}, \ldots, \mathbf{U}^{(N)}]\!].$$

In other words, mode-n matrix multiplication just modifies the nth factor matrix in the Kruskal tensor. The work is just a matrix-matrix multiply, $O(RI_nJ)$. More generally, if $\mathbf{V}^{(n)}$ is of size $J_n \times I_n$ for $n = 1, \ldots, N$, then

$$[\![\mathcal{X} \,; \mathbf{V}^{(1)}, \ldots, \mathbf{V}^{(N)}]\!] = [\![\boldsymbol{\lambda} \,; \mathbf{V}^{(1)}\mathbf{U}^{(1)}, \ldots, \mathbf{V}^{(N)}\mathbf{U}^{(N)}]\!],$$

retains the Kruskal tensor format and the work is N matrix-matrix multiplies for $O(R \sum_n I_n J_n)$.

5.2.3 Mode-n vector multiplication for a Kruskal tensor

In multiplication of a Kruskal tensor by a vector, the nth factor matrix necessarily disappears and is absorbed into the weights. Let $\mathbf{v} \in \mathbb{R}^{I_n}$, then

$$\mathcal{X} \,\bar{\times}_n\, \mathbf{v} = [\![\boldsymbol{\lambda} * \mathbf{w} \,; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(n-1)}, \mathbf{U}^{(n+1)}, \ldots, \mathbf{U}^{(N)}]\!] \quad \text{where} \quad \mathbf{w} = \mathbf{U}^{(n)\mathsf{T}} \mathbf{v}.$$

This operation retains the Kruskal tensor structure (though its order is reduced), and the work is multiplying a matrix times a vector and then a Hadamard product of two vectors, i.e., $O(RI_n)$. More generally, multiplying a Kruskal tensor by a vector $\mathbf{v}^{(n)} \in \mathbb{R}^{I_n}$ in every mode yields:

$$\mathcal{X} \bar{\times}_1 \mathbf{v}^{(1)} \bar{\times}_2 \mathbf{v}^{(2)} \cdots \bar{\times}_N \mathbf{v}^{(N)} = \boldsymbol{\lambda}^{\mathsf{T}} \left(\mathbf{w}^{(1)} * \mathbf{w}^{(2)} * \cdots * \mathbf{w}^{(N)} \right)$$
$$\text{where} \quad \mathbf{w}^{(n)} = \mathbf{U}^{(n)\mathsf{T}} \mathbf{v}^{(n)} \text{ for all } n = 1, \ldots, N.$$

Here, the final result is a scalar, which is computed by N matrix-vector products, N vector Hadamard products, and one vector dot-product, for total work of $O(R \sum_n I_n)$.

5.2.4 Inner product of two Kruskal tensors

Consider Kruskal tensors $\mathcal{X}$ and $\mathcal{Y}$, both of size $I_1 \times I_2 \times \cdots \times I_N$, given by:

$$\mathcal{X} = [\![\boldsymbol{\lambda} \,; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!] \quad \text{and} \quad \mathcal{Y} = [\![\boldsymbol{\sigma} \,; \mathbf{V}^{(1)}, \ldots, \mathbf{V}^{(N)}]\!].$$

Assume that $\mathcal{X}$ has R rank-1 factors and $\mathcal{Y}$ has S. From (16), we have

$$\begin{aligned} \langle \mathcal{X}, \mathcal{Y} \rangle &= \langle \operatorname{vec}(\mathcal{X}), \operatorname{vec}(\mathcal{Y}) \rangle \\ &= \boldsymbol{\lambda}^{\mathsf{T}} \left(\mathbf{U}^{(N)} \odot \cdots \odot \mathbf{U}^{(1)} \right)^{\mathsf{T}} \left(\mathbf{V}^{(N)} \odot \cdots \odot \mathbf{V}^{(1)} \right) \boldsymbol{\sigma} \\ &= \boldsymbol{\lambda}^{\mathsf{T}} \left(\mathbf{U}^{(N)\mathsf{T}} \mathbf{V}^{(N)} * \cdots * \mathbf{U}^{(1)\mathsf{T}} \mathbf{V}^{(1)} \right) \boldsymbol{\sigma}. \end{aligned}$$

Note that this does not require that the number of rank-1 factors in $\mathcal{X}$ and $\mathcal{Y}$ to be the same. The work is N matrix-matrix multiplies, plus N Hadamard products, and a final vector-matrix-vector product. The total work is $O(RS \sum_n I_n)$.

5.2.5 Norm of a Kruskal tensor

Let $\mathcal{X}$ be a Kruskal tensor as defined in (14). From §5.2.4, it follows directly that

$$\| \mathcal{X} \|^2 = \langle \mathcal{X}, \mathcal{X} \rangle = \boldsymbol{\lambda}^{\mathsf{T}} \left(\mathbf{U}^{(N)\mathsf{T}} \mathbf{U}^{(N)} * \cdots * \mathbf{U}^{(1)\mathsf{T}} \mathbf{U}^{(1)} \right) \boldsymbol{\lambda},$$

and the total work is $O(R^2 \sum_n I_n)$.

5.2.6 Matricized Kruskal tensor times Khatri-Rao product

As noted in §2.6, a common operation is to calculate (6). Let $\mathcal{X}$ be a Kruskal tensor as in (14). And, let $\mathbf{V}^{(m)}$ be of size $I_m \times S$ for $m \neq n$. In the case of a Kruskal tensor, the operation simplifies to:

$$\begin{aligned} \mathbf{W} &= \mathcal{X} \left(\mathbf{V}^{(N)} \odot \cdots \odot \mathbf{V}^{(n+1)} \odot \mathbf{V}^{(n-1)} \odot \cdots \odot \mathbf{V}^{(1)} \right) \\ &= \mathbf{U}^{(n)} \boldsymbol{\Lambda} \left(\mathbf{U}^{(N)} \odot \cdots \odot \mathbf{U}^{(n+1)} \odot \mathbf{U}^{(n-1)} \odot \cdots \odot \mathbf{U}^{(1)} \right)^{\mathsf{T}} \\ &\qquad \left(\mathbf{V}^{(N)} \odot \cdots \odot \mathbf{V}^{(n+1)} \odot \mathbf{V}^{(n-1)} \odot \cdots \odot \mathbf{V}^{(1)} \right). \end{aligned}$$

Using the properties of the Khatri-Rao product [42] and setting $\mathbf{A}^{(m)} = \mathbf{U}^{(m)\mathsf{T}}\mathbf{V}^{(m)} \in \mathbb{R}^{R\times S}$ for all $m \neq n$, we have

$$\mathbf{W} = \mathbf{U}^{(n)}\mathbf{\Lambda}\left(\mathbf{A}^{(N)} * \cdots * \mathbf{A}^{(n+1)} * \mathbf{A}^{(n-1)} * \cdots * \mathbf{A}^{(1)}\right).$$

Computing each $\mathbf{A}^{(m)}$ requires a matrix-matrix product for a cost of $O(RSI_m)$ for each $m = 1,\ldots,n-1,n+1,\ldots,N$. There is also a sequence of $N-1$ Hadamard products of $R\times S$ matrices, multiplication with an $R\times R$ diagonal matrix, and finally matrix-matrix multiplication that costs $O(RSI_n)$. Thus, the total cost is $O(RS\sum_n I_n)$.

5.2.7 Computing $\mathbf{X}_{(n)}\mathbf{X}_{(n)}^{\mathsf{T}}$

Let $\mathcal{X}$ be a Kruskal tensor as in (14). We can use the properties of the Khatri-Rao product to efficiently compute

$$\mathbf{Z} = \mathbf{X}^{(n)}\mathbf{X}^{(n)\mathsf{T}} \in \mathbb{R}^{I_n\times I_n}.$$

From (11),

$$\mathbf{Z} = \mathbf{U}^{(n)}\mathbf{\Lambda}\left(\mathbf{U}^{(N)} \odot \cdots \odot \mathbf{U}^{(n+1)} \odot \mathbf{U}^{(n-1)} \odot \cdots \odot \mathbf{U}^{(1)}\right)^{\mathsf{T}} \\ \left(\mathbf{U}^{(N)} \odot \cdots \odot \mathbf{U}^{(n+1)} \odot \mathbf{U}^{(n-1)} \odot \cdots \odot \mathbf{U}^{(1)}\right)\mathbf{\Lambda}\mathbf{U}^{(n)\mathsf{T}}.$$

This reduces to

$$\mathbf{Z} = \mathbf{U}^{(n)}\mathbf{\Lambda}\left(\mathbf{V}^{(N)} * \cdots * \mathbf{V}^{(n+1)} * \mathbf{V}^{(n-1)} * \cdots * \mathbf{V}^{(1)}\right)\mathbf{\Lambda}\mathbf{U}^{(n)\mathsf{T}},$$

where $\mathbf{V}^{(m)} = \mathbf{U}^{(m)\mathsf{T}}\mathbf{U}^{(m)} \in \mathbb{R}^{R\times R}$ for all $m \neq n$ and costs $O(R^2 I_m)$. This is followed by $(N-1)$ $R\times R$ matrix Hadamard products, and two matrix multiplies. The total work in $O(R^2\sum_n I_n)$.

5.3 MATLAB details for Kruskal tensors

A Kruskal tensor $\mathcal{X}$ from (14) is constructed in MATLAB by passing in the matrices $\mathbf{U}^{(1)},\ldots,\mathbf{U}^{(N)}$ and the weighting vector $\boldsymbol{\lambda}$ using `X = ktensor(lambda,{U1,U2,U3})`. If all the $\boldsymbol{\lambda}$-values are one, then the shortcut `X = ktensor({U1,U2,U3})` can be used instead. In version 1.0 of the Tensor Toolbox, this object was called the `cp_tensor` [4].

A Kruskal tensor can be converted to a standard `tensor` by calling `full(X)`. Subscripted reference and assignment can only be done on the component matrices, not elementwise. For example, it is possible to change the 4th element of $\boldsymbol{\lambda}$ but not the $(1,1,1)$ element of a three-way Kruskal tensor $\mathcal{X}$. Scalar multiplication is supported, i.e., `X*5`. It is also possible to add to Kruskal tensors (`X+Y` or `X-Y`) as described in §5.2.1.

The n-mode product of a Kruskal tensor with one or more matrices (§5.2.2) or vectors (§5.2.3) is implemented in `ttm` and `ttv`, respectively. The inner product (§5.2.4 and also §6) is called via `innerprod`. The norm of a Kruskal tensor (§5.2.5) is computed by calling `norm`. The function `mttkrp` computes the matricized-tensor-times-Khatri-Rao-product as described in §5.2.6. The function `nvecs(X,n)` computes the leading mode-n eigenvectors for $\mathbf{X}_{(n)}\mathbf{X}_{(n)}^{\mathsf{T}}$ as described in §5.2.7.

This page intentionally left blank.

6 Operations that combine different types of tensors

Here we consider two operations that combine different types of tensors. Throughout, we work with the following tensors:

- $\mathcal{D}$ is a dense tensor of size $I_1 \times I_2 \times \cdots \times I_N$.

- $\mathcal{S}$ is a sparse tensor of size $I_1 \times I_2 \times \cdots \times I_N$, and $\mathbf{v} \in \mathbb{R}^P$ contains its nonzeros.

- $\mathcal{T} = [\![\mathcal{G} \,; \mathbf{U}^{(1)}, \ldots, \mathbf{U}^{(N)}]\!]$ is a Tucker tensor of size $I_1 \times I_2 \times \cdots \times I_N$ with a core of size $\mathcal{G} \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times J_N}$ and factor matrices $\mathbf{U}^{(n)} \in \mathbb{R}^{I_n \times J_n}$ for all n.

- $\mathcal{K} = [\![\boldsymbol{\lambda} \,; \mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(N)}]\!]$ is a Kruskal tensor of size $I_1 \times I_2 \times \cdots \times I_N$ and R factor matrices $\mathbf{W}^{(n)} \in \mathbb{R}^{I_n \times R}$.

6.1 Inner Product

Here, we discuss how to compute the inner product between any pair of tensors of different types.

For a sparse and dense tensor, we have $\langle \mathcal{D}, \mathcal{S} \rangle = \mathbf{v}^{\mathsf{T}}\mathbf{z}$, where $\mathbf{z}$ is the vector extracted from $\mathcal{D}$ using the indices of the nonzeros in the sparse tensor $\mathcal{S}$.

For a Tucker and dense tensor, if the core of the Tucker tensor is small, we can compute

$$\langle \mathcal{T}, \mathcal{D} \rangle = \langle \mathcal{G}, \hat{\mathcal{D}} \rangle \quad \text{where} \quad \hat{\mathcal{D}} = \mathcal{D} \times_1 \mathbf{U}^{(1)\mathsf{T}} \cdots \times_n \mathbf{U}^{(N)\mathsf{T}}.$$

Computing $\hat{\mathcal{D}}$ and its inner product with a dense $\mathcal{G}$ costs

$$O\left(\sum_{n=1}^{N} \left(\prod_{p=n}^{N} I_p \prod_{q=1}^{n} J_q \right) + \prod_{n=1}^{N} J_n \right).$$

The procedure is the same for a Tucker tensor and a sparse tensor, i.e., $\langle \mathcal{T}, \mathcal{S} \rangle$, though the cost is different (see §3.2.5).

For the inner product of a Kruskal tensor and a dense tensor, we have

$$\langle \mathcal{D}, \mathcal{K} \rangle = \text{vec}(\mathcal{D})^{\mathsf{T}} \left(\mathbf{U}^{(N)} \odot \cdots \odot \mathbf{U}^{(1)} \right) \boldsymbol{\lambda}.$$

The cost of forming the Khatri-Rao product dominates: $O(R \prod_n I_n)$.

The inner product of a Kruskal tensor and a sparse tensor can be written as

$$\langle \mathcal{S}, \mathcal{K} \rangle = \sum_{r=1}^{R} \lambda_r (\mathcal{S} \,\bar{\times}_1\, \mathbf{w}_r^{(1)} \cdots \,\bar{\times}_N\, \mathbf{w}_r^{(N)}).$$

Consequently, the cost is equivalent to doing R tensor-times-vector products with N vectors each, i.e., $O(RN \cdot \text{nnz}(\mathcal{S}))$, The same reasoning applies to the inner product of Tucker and Kruskal tensors, $\langle \mathcal{T}, \mathcal{K} \rangle$.

6.2 Hadamard product

We consider the Hadamard product of a sparse tensor with dense and Kruskal tensors.

The product $\mathcal{Y} = \mathcal{D} * \mathcal{S}$ necessarily has zeros everywhere that $\mathcal{S}$ is zero, so only the potential nonzeros in the result, corresponding to the nonzeros in $\mathcal{S}$, need to be computed. The result is assembled from the nonzero subscripts of S and $\mathbf{v} * \mathbf{z}$, where $\mathbf{z}$ is the values of $\mathcal{D}$ at the nonzero subscripts of S. The work is $O(\text{nnz}(\mathcal{S}))$.

Once again, $\mathcal{Y} = \mathcal{S} * \mathcal{K}$ can only have nonzeros where $\mathcal{S}$ has nonzeros. Let $\mathbf{z} \in \mathbb{R}^P$ be the vector of possible nonzeros for $\mathcal{Y}$ corresponding to the locations of the nonzeros in $\mathcal{S}$. Observe that

$$z_p = v_p \left(\sum_{r=1}^{R} \lambda_r w^{(1)}_{r,s_{1p}} w^{(2)}_{r,s_{2p}} \cdots w^{(N)}_{r,s_{Np}} \right).$$

This means that we can compute it vectorwise by a sum of a series of vector Hadamard products with "expanded" vectors as in §3.2.4, for example. The work is $O(N \cdot \text{nnz}(\mathcal{S}))$.

7 Conclusions

In this article, we considered the question of how to deal with potentially large-scale tensors stored in sparse or factored (Tucker or Kruskal) form. The Tucker and Kruskal formats can be used, for example, to store the results of a Tucker or CANDECOMP/PARAFAC decomposition of a large, sparse tensor. We demonstrated relevant mathematical properties of structured tensors that simplify common operations appearing in tensor decomposition algorithms, such as mode-n matrix/vector multiplication, inner product, and collapsing/scaling. For many functions, we are able to realize substantial computational efficiencies as compared to working with the tensors in dense/unfactored form.

The Tensor Toolbox provides an extension to MATLAB by adding the ability to work with sparse multi-dimensional arrays, not to mention the specialized factored tensors. Moreover, relatively few packages in any language have the ability to work with sparse tensors, and our investigations have not revealed any others that have the variety of capabilities available in the Tensor Toolbox. A complete listing of functions for dense (`tensor`), sparse (`sptensor`), Tucker (`ttensor`), and Kruskal (`ktensor`) tensors is provided in Table 1. In general, Tensor Toolbox objects work the same as MATLAB arrays. For example, for a 3-way tensor $\mathcal{A}$ in any format (`tensor`, `sptensor`, `ktensor`, `ttensor`), it is possible to call functions such as `size(A)`, `ndims(A)`, `permute(A,[3 2 1])`, `-A`, `2*A`, `norm(A)` (always the Frobenius norm for tensors). A major difference between Tensor Toolbox objects and MATLAB arrays is that the tensor classes support subscript indexing (i.e., passing in a matrix of subscripts) and do not support linear indexing. This avoids possible complications with integer overflow for large-scale arrays; see §3.3.

Due to their structure, factored tensors cannot support every operation that is supported for dense and sparse tensors. For instance, most element-level operations are prohibited, such as subscripted reference/assignment, logical operations/comparisons, etc. In these cases, memory permitting, the factored tensors can be converted to dense tensors by calling `full`. However, there are certain operations that can be adapted to the structure. For example, it is possible to add two Kruskal tensors, as described in §5.2.1, and it is possible to do tensor multiplication and inner products involving Kruskal tensors, see §6.

A major feature of the Tensor Toolbox is that it defines multiplication on tensor objects. For example, generalized tensor-tensor multiplication and contraction is supported for dense and sparse tensors. The specialized operations of n-mode multiplication of a tensor by a matrix or a vector is supported for dense, sparse, and factored tensors. Likewise, inner products, even between tensors of different types, and norms are supported across the board.

The Tensor Toolbox also includes specialized functions, such as `collapse` and `scale` (see §3.2.9), the matricized tensor times Khatri-Rao product (see §2.6), the

Functionality	`tensor`	`sptensor`	`ttensor`	`ktensor`
Sizes (`size, ndims`)	✓	✓	✓	✓
Number of nonzeros (`nnz`)	✓[d]	✓	–	–
Permute	✓	✓	✓	✓
Remove singleton dimensions (`squeeze`)	✓	✓	–	–
Subscripted Reference & Assignment	✓[a]	✓[a]	✓[b]	✓[b]
Unary plus and minus (e.g., `-X`)	✓	✓	✓	✓
Plus and minus	✓	✓	–	✓
Logical (`and/or/xor/not`)	✓	✓[d]	–	–
Comparisons (`eq/ne/gt/ge/lt/le`)	✓	✓[d]	–	–
Scalar multiplication (`A*5`)	✓	✓	✓	✓
Scalar elementwise power (`A.^5`)	✓	✓[d]	–	✓
Array (Hadamard) multiplication (`A.*B`)	✓[c]	✓[c]	–	✓[c]
Array right division (`A./B`)	✓[c]	✓[c]	–	✓[c]
Convert to MDA (`double`)	✓	✓	✓	✓
Convert to dense (`full`)	✓	✓	✓	✓
Find subscripts of nonzero elements (`find`)	✓	✓	–	–
Apply a function to every element (`tenfun`)	✓	–	–	–
Apply a function to every nonzero element (`elemfun`)	–	✓	–	–
Tensor times tensor (`ttt`)	✓	✓	–	–
Generalized trace (`contract`)	✓	✓	–	–
Tensor time matrix (`ttm`)	✓	✓	✓	✓
Tensor times vector (`ttv`)	✓	✓	✓	✓
Matricized tensor times Khatri-Rao product (`mttkrp`)	✓	✓	✓	✓
Mode-n singular vectors (`nvecs`)	✓	✓	✓	✓
Inner product (`innerprod`)	✓[c]	✓[c]	✓[c]	✓[c]
Norm	✓	✓	✓	✓
Collapse along dimensions	✓	✓	–	–
Scale along dimensions	✓	✓	–	–
Matricize	✓	✓	–	–

[a] Multiple subscripts passed explicitly (no linear indices).

[b] Only the factors may be referenced/modified.

[c] Supports combinations of different types of tensors.

[d] New as of version 2.1.

Table 1. Methods in the Tensor Toolbox

computation of the leading mode-n singular vectors (equivalent to the leading eigenvectors of $\mathbf{X}_{(n)}\mathbf{X}_{(n)}^{\mathsf{T}}$), and conversion of a tensor to a matrix.

While we believe that the Tensor Toolbox is a useful package, we look forward to greater availability of storage formats and increased functionality in software for tensors, especially sparse tensors. For instance, the benefits of storing matrices in sorted order using CSR or CSC format generally outweigh the negatives, and so it makes sense to seek multidimensional extensions that are both practical and useful, at least for specialized contexts as with the EKMR [32, 33].

Furthermore, extensions to parallel data structures and architectures requires further innovation, especially as we hope to leverage existing codes for parallel linear algebra.

References

[1] E. ACAR, S. A. ÇAMTEPE, AND B. YENER, *Collective sampling and analysis of high order tensors for chatroom communications*, in ISI 2006: IEEE International Conference on Intelligence and Security Informatics, vol. 3975 of Lecture Notes in Computer Science, Berlin, 2006, Springer, pp. 213–224.

[2] C. A. ANDERSSON AND R. BRO, *The N-way toolbox for MATLAB*, Chemometr. Intell. Lab., 52 (2000), pp. 1–4. See also `http://www.models.kvl.dk/source/nwaytoolbox/`.

[3] C. J. APPELLOF AND E. R. DAVIDSON, *Strategies for analyzing data from video fluorometric monitoring of liquid chromatographic effluents*, Anal. Chem., 53 (1981), pp. 2053–2056.

[4] B. W. BADER AND T. G. KOLDA, *MATLAB tensor classes for fast algorithm prototyping*, Tech. Report SAND2004-5187, Sandia National Laboratories, Albuquerque, New Mexico and Livermore, California, Oct. 2004. To appear in ACM Trans. Math. Softw.

[5] ——, *Matlab tensor toolbox, version 2.1*. `http://csmr.ca.sandia.gov/~tgkolda/TensorToolbox/`, December 2006.

[6] R. BRO, *PARAFAC. tutorial and applications*, Chemometr. Intell. Lab., 38 (1997), pp. 149–171.

[7] ——, *Multi-way analysis in the food industry: models, algorithms, and applications*, PhD thesis, University of Amsterdam, 1998. Available at `http://www.models.kvl.dk/research/theses/`.

[8] J. D. CARROLL AND J. J. CHANG, *Analysis of individual differences in multidimensional scaling via an N-way generalization of 'Eckart-Young' decomposition*, Psychometrika, 35 (1970), pp. 283–319.

[9] B. CHEN, A. PETROPOLU, AND L. DE LATHAUWER, *Blind identification of convolutive MIM systems with 3 sources and 2 sensors*, Applied Signal Processing, (2002), pp. 487–496. (Special Issue on Space-Time Coding and Its Applications, Part II).

[10] P. COMON, *Tensor decompositions: state of the art and applications*, in Mathematics in Signal Processing V, J. G. McWhirter and I. K. Proudler, eds., Oxford University Press, Oxford, UK, 2001, pp. 1–24.

[11] L. DE LATHAUWER, B. DE MOOR, AND J. VANDEWALLE, *A multilinear singular value decomposition*, SIAM J. Matrix Anal. A., 21 (2000), pp. 1253–1278.

[12] ——, *On the best rank-1 and rank-*$(R_1, R_2, \ldots, R_N)$ *approximation of higher-order tensors*, SIAM J. Matrix Anal. A., 21 (2000), pp. 1324–1342.

[13] I. S. Duff and J. K. Reid, *Some design features of a sparse matrix code*, ACM Trans. Math. Softw., 5 (1979), pp. 18–35.

[14] R. Garcia and A. Lumsdaine, *MultiArray: a C++ library for generic programming with arrays*, Software: Practice and Experience, 35 (2004), pp. 159–188.

[15] J. R. Gilbert, C. Moler, and R. Schreiber, *Sparse matrices in MATLAB: design and implementation*, SIAM J. Matrix Anal. A., 13 (1992), pp. 333–356.

[16] V. S. Grigorascu and P. A. Regalia, *Tensor displacement structures and polyspectral matching*, in Fast Reliable Algorithms for Matrices with Structure, T. Kaliath and A. H. Sayed, eds., SIAM, Philadelphia, 1999, pp. 245–276.

[17] F. G. Gustavson, *Some basic techniques for solving sparse systems*, in Sparse Matrices and their Applications, D. J. Rose and R. A. Willoughby, eds., Plenum Press, New York, 1972, pp. 41–52.

[18] R. A. Harshman, *Foundations of the PARAFAC procedure: models and conditions for an "explanatory" multi-modal factor analysis*, UCLA working papers in phonetics, 16 (1970), pp. 1–84. Available at `http://publish.uwo.ca/~harshman/wpppfac0.pdf`.

[19] R. Henrion, *Body diagonalization of core matrices in three-way principal components analysis: Theoretical bounds and simulation*, J. Chemometr., 7 (1993), pp. 477–494.

[20] ——, *N-way principal component analysis theory, algorithms and applications*, Chemometr. Intell. Lab., 25 (1994), pp. 1–23.

[21] H. A. Kiers, *Joint orthomax rotation of the core and component matrices resulting from three-mode principal components analysis*, J. Classif., 15 (1998), pp. 245 – 263.

[22] H. A. L. Kiers, *Towards a standardized notation and terminology in multiway analysis*, J. Chemometr., 14 (2000), pp. 105–122.

[23] T. G. Kolda, *Orthogonal tensor decompositions*, SIAM J. Matrix Anal. A., 23 (2001), pp. 243–255.

[24] ——, *Multilinear operators for higher-order decompositions*, Tech. Report SAND2006-2081, Sandia National Laboratories, Albuquerque, New Mexico and Livermore, California, Apr. 2006.

[25] P. Kroonenberg, *Applications of three-mode techniques: overview, problems, and prospects (slides)*. Presentation at the AIM Tensor Decompositions Workshop, Palo Alto, California, July 2004. Available at `http://csmr.ca.sandia.gov/~tgkolda/tdw2004/Kroonenberg%20-%20Talk.pdf`.

[26] P. M. Kroonenberg and J. De Leeuw, *Principal component analysis of three-mode data by means of alternating least squares algorithms*, Psychometrika, 45 (1980), pp. 69–97.

[27] J. B. Kruskal, *Three-way arrays: rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics*, Linear Algebra Appl., 18 (1977), pp. 95–138.

[28] J. B. Kruskal, *Rank, decomposition, and uniqueness for 3-way and N-way arrays*, in Multiway Data Analysis, R. Coppi and S. Bolasco, eds., North-Holland, Amsterdam, 1989.

[29] W. Landry, *Implementing a high performance tensor library*, Scientific Programming, 11 (2003), pp. 273–290.

[30] S. Leurgans and R. T. Ross, *Multilinear models: applications in spectroscopy*, Stat. Sci., 7 (1992), pp. 289–310.

[31] L.-H. Lim, *Singular values and eigenvalues of tensors: a variational approach*, in CAMAP2005: 1st IEEE International Workshop on Computational Advances in Multi-Sensor Adaptive Processing, 2005, pp. 129–132.

[32] C.-Y. Lin, Y.-C. Chung, and J.-S. Liu, *Efficient data compression methods for multidimensional sparse array operations based on the ekmr scheme*, IEEE Transactions on Computers, 52 (2003), pp. 1640–1646.

[33] C.-Y. Lin, J.-S. Liu, and Y.-C. Chung, *Efficient representation scheme for multidimensional array operations*, IEEE Transactions on Computers, 51 (2002), pp. 327–345.

[34] R. P. McDonald, *A simple comprehensive model for the analysis of covariance structures*, Brit. J. Math. Stat. Psy., 33 (1980), p. 161. Cited in [7].

[35] M. Mørup, L. Hansen, J. Parnas, and S. M. Arnfred, *Decomposing the time-frequency representation of EEG using nonnegative matrix and multiway factorization*. Available at `http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/4144/pdf/imm4144.pdf`, 2006.

[36] P. Paatero, *The multilinear engine - a table-driven, least squares program for solving multilinear problems, including the n-way parallel factor analysis model*, J. Comput. Graph. Stat., 8 (1999), pp. 854–888.

[37] U. W. Pooch and A. Nieder, *A survey of indexing techniques for sparse matrices*, ACM Computing Surveys, 5 (1973), pp. 109–133.

[38] C. R. Rao and S. Mitra, *Generalized inverse of matrices and its applications*, Wiley, New York, 1971. Cited in [7].

[39] J. R. Ruíz-Tolosa and E. Castillo, *From vectors to tensors*, Universitext, Springer, Berlin, 2005.

[40] Y. Saad, *Iterative Methods for Sparse Linear Systems, Second Edition*, SIAM, Philadelphia, 2003.

[41] B. Savas, *Analyses and tests of handwritten digit recognition algorithms*, master's thesis, Linköping University, Sweden, 2003.

[42] A. Smilde, R. Bro, and P. Geladi, *Multi-way analysis: applications in the chemical sciences*, Wiley, West Sussex, England, 2004.

[43] J. Sun, D. Tao, and C. Faloutsos, *Beyond streams and graphs: dynamic tensor analysis*, in KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, 2006, pp. 374–383.

[44] J.-T. Sun, H.-J. Zeng, H. Liu, Y. Lu, and Z. Chen, *CubeSVD: a novel approach to personalized Web search*, in WWW 2005: Proceedings of the 14th international conference on World Wide Web, ACM Press, New York, 2005, pp. 382–390.

[45] J. Ten Berge, J. De Leeuw, and P. M. Kroonenberg, *Some additional results on principal components analysis of three-mode data by means of alternating least squares algorithms*, Psychometrika, 52 (1987), pp. 183–191.

[46] J. M. F. Ten Berge and H. A. L. Kiers, *Simplicity of core arrays in three-way principal component analysis and the typical rank of* $p \times q \times 2$ *arrays*, Linear Algebra Appl., 294 (1999), pp. 169–179.

[47] G. Tomasi, *Use of the properties of the Khatri-Rao product for the computation of Jacobian, Hessian, and gradient of the PARAFAC model under MATLAB*, 2005.

[48] G. Tomasi and R. Bro, *A comparison of algorithms for fitting the PARAFAC model*, Comput. Stat. Data. An., (2005).

[49] L. R. Tucker, *Some mathematical notes on three-mode factor analysis*, Psychometrika, 31 (1966), pp. 279–311.

[50] M. A. O. Vasilescu and D. Terzopoulos, *Multilinear analysis of image ensembles: TensorFaces*, in ECCV 2002: 7th European Conference on Computer Vision, vol. 2350 of Lecture Notes in Computer Science, Springer, Berlin, 2002, pp. 447–460.

[51] D. Vlasic, M. Brand, H. Pfister, and J. Popović, *Face transfer with multilinear models*, ACM Transactions on Graphics, 24 (2005), pp. 426–433. Proceedings of ACM SIGGRAPH 2005.

[52] H. Wang and N. Ahuja, *Facial expression decomposition*, in ICCV 2003: 9th IEEE International Conference on Computer Vision, vol. 2, 2003, pp. 958–965.

[53] R. Zass, *HUJI tensor library*. `http://www.cs.huji.ac.il/~zass/htl/`, May 2006.

[54] E. Zhang, J. Hays, and G. Turk, *Interactive tensor field design and visualization on surfaces*. Available at `http://eecs.oregonstate.edu/library/files/2005-106/tenflddesn.pdf`, 2005.

[55] T. Zhang and G. H. Golub, *Rank-one approximation to high order tensors*, SIAM J. Matrix Anal. A., 23 (2001), pp. 534–550.

DISTRIBUTION:

1 Evrim Acar (acare@rpi.edu)
Department of Computer Science, Rensselaer Polytechnic Institute, USA

1 Professor Rasmus Bro (rb@kvl.dk)
Chemometrics Group, Department of Food Science, The Royal Veterinary and Agricultural University (KVL), Denmark

1 Professor Petros Drineas (drinep@cs.rpi.edu)
Department of Computer Science, Rensselaer Polytechnic Institute, USA

1 Professor Lars Eldén (laeld@liu.se)
Department of Mathematics, Linköping University, Sweden

1 Professor Christos Faloutsos (christos@cs.cmu.edu)
Department of Computer Science, Carnegie Mellon University, USA

1 Derry FitzGerald (derry.fitzgerald@cit.ie)
Cork Institute of Technology, Ireland

1 Professor Michael Friedlander (mpf@cs.ubc.ca)
Department of Computer Science, University of British Columbia, Canada

1 Professor Gene Golub (golub@stanford.edu)
Stanford University, USA

1 Jerry Gregoire (jgregoire@ece.montana.edu)
Montana State University, USA

1 Professor Richard Harshman (harshman@uwo.ca)
Department of Psychology, University of Western Ontario, Canada

1 Professor Henk Kiers (h.a.l.kiers@rug.nl)
Heymans Institute, University of Groningen, The Netherlands

1 Professor Misha Kilmer (misha.kilmer@tufts.edu)
Department of Mathematics, Tufts University, Boston, USA

1 Professor Pieter Kroonenberg (kroonenb@fsw.leidenuniv.nl)
Department of Education and Child Studies, Leiden University, The Netherlands

1 Walter Landry (wlandry@ucsd.edu)
University of California, San Diego, USA

1 Lieven De Lathauwer (Lieven.DeLathauwer@ensea.fr)
ENSEA, France

1 Lek-Heng Lim (lekheng@math.Stanford.edu)
Stanford University, USA

1 Michael Mahoney (mahoney@yahoo-inc.com)
Yahoo! Research Labs, USA

1 Morten Mørup (morten.morup@gmail.com)
Department of Intelligent Signal Processing, Technical University of Denmark, Denmark

1 Professor Dianne O'Leary (oleary@cs.umd.edu)
Department of Computer Science, University of Maryland, USA

1 Professor Pentti Paatero (Pentti.Paatero@Helsinki.fi)
Department of Physics, University of Helsinki, Finland

1 Berkant Savas (besav@mai.liu.se)
Department of Mathematics, Linköping University, Sweden

1 Jimeng Sun (jimeng@cs.cmu.edu)
Department of Computer Science, Carnegie Mellon University, USA

1 Professor Jos Ten Berge (J.M.F.ten.Berge@rug.nl)
Heijmans Instituut Rijksuniversiteit Groningen, The Netherlands

1 Giorgio Tomasi (giorgio.tomasi@gmail.com)
The Royal Veterinary and Agricultural University (KVL), Denmark

1 Professor Bülent Yener (yener@cs.rpi.edu)
Department of Computer Science, Rensselaer Polytechnic Institute, USA

1 Ron Zass (zass@cs.huji.ac.il)
Computer Vision Lab, School of Computer Science and Engineering, The Hebrew University of Jerusalem, Israel

5	MS 1318	Brett Bader, 1416
1	MS 1318	Andrew Salinger, 1416
1	MS 9159	Heidi Ammerlahn, 8962
5	MS 9159	Tammy Kolda, 8962
1	MS 9915	Craig Smith, 8529
2	MS 0899	Technical Library, 4536
2	MS 9018	Central Technical Files, 8944
1	MS 0323	Donna Chavez, LDRD Office, 1011