Empirical Evaluation of Four Tensor Decomposition Algorithms

Peter D. Turney
Institute for Information Technology
National Research Council of Canada
M-50 Montreal Road
Ottawa, Ontario, Canada
K1A 0R6
peter.turney@nrc-cnrc.gc.ca

Technical Report ERB-1152, NRC-49877
November 12, 2007

Abstract

Higher-order tensor decompositions are analogous to the familiar Singular Value Decomposition (SVD), but they transcend the limitations of matrices (second-order tensors). SVD is a powerful tool that has achieved impressive results in information retrieval, collaborative filtering, computational linguistics, computational vision, and other fields. However, SVD is limited to two-dimensional arrays of data (two modes), and many potential applications have three or more modes, which require higher-order tensor decompositions. This paper evaluates four algorithms for higher-order tensor decomposition: Higher-Order Singular Value Decomposition (HO-SVD), Higher-Order Orthogonal Iteration (HOOI), Slice Projection (SP), and Multislice Projection (MP). We measure the time (elapsed run time), space (RAM and disk space requirements), and fit (tensor reconstruction accuracy) of the four algorithms, under a variety of conditions. We find that standard implementations of HO-SVD and HOOI do not scale up to larger tensors, due to increasing RAM requirements. We recommend HOOI for tensors that are small enough for the available RAM and MP for larger tensors.

1 Introduction

Singular Value Decomposition (SVD) is growing increasingly popular as a tool for the analysis of two-dimensional arrays of data, due to its success in a wide variety of applications, such as information retrieval (Deerwester et al., 1990), collaborative filtering (Billsus and Pazzani, 1998), computational linguistics (Schütze, 1998), computational vision (Brand, 2002), and genomics (Alter et al., 2000). SVD is limited to two-dimensional arrays (matrices or second-order tensors), but many applications require higher-dimensional arrays, known as higher-order tensors.

There are several higher-order tensor decompositions, analogous to SVD, that are able to capture higher-order structure that cannot be modeled with two dimensions (two modes). Higher-order generalizations of SVD include Higher-Order Singular Value Decomposition (HO-SVD) (De Lathauwer et al., 2000a), Tucker decomposition (Tucker, 1966), and PARAFAC (parallel factor analysis) (Harshman, 1970), which is also known as CANDECOMP (canonical decomposition) (Carroll and Chang, 1970).

Higher-order tensors quickly become unwieldy. The number of elements in a matrix increases quadratically, as the product of the number of rows and columns, but the number of elements in a third-order tensor increases cubically, as a product of the number of rows, columns, and tubes. Thus there is a need for tensor decomposition algorithms that can handle large tensors.

In this paper, we evaluate four algorithms for higher-order tensor decomposition: Higher-Order Singular Value Decomposition (HO-SVD) (De Lathauwer et al., 2000a), Higher-Order Orthogonal Iteration (HOOI) (De Lathauwer et al., 2000b), Slice Projection (SP) (Wang and Ahuja, 2005), and Multislice Projection (MP) (introduced here). Our main concern is the ability of the four algorithms to scale up to large tensors.

In Section 2, we motivate this work by listing some of the applications for higher-order tensors. In any field where SVD has been useful, there is likely to be a third or fourth mode that has been ignored, because SVD only handles two modes.

The tensor notation we use in this paper is pre-

sented in Section 3. We follow the notational conventions of Kolda (2006).

Section 4 presents the four algorithms, HO-SVD, HOOI, SP, and MP. For HO-SVD and HOOI, we used the implementations given in the MATLAB Tensor Toolbox (Bader and Kolda, 2007a; Bader and Kolda, 2007b). For SP and MP, we created our own MATLAB implementations. Our implementation of MP for third-order tensors is given in the Appendix.

Section 5 presents our empirical evaluation of the four tensor decomposition algorithms. In the experiments, we measure the time (elapsed run time), space (RAM and disk space requirements), and fit (tensor reconstruction accuracy) of the four algorithms, under a variety of conditions.

The first group of experiments looks at how the algorithms scale as the input tensors grow increasingly larger. We test the algorithms with random sparse third-order tensors as input. HO-SVD and HOOI exceed the available RAM when given larger tensors as input, but SP and MP are able to process large tensors with low RAM usage and good speed. HOOI provides the best fit, followed by MP, then SP, and lastly HO-SVD.

The second group of experiments examines the sensitivity of the fit to the balance in the ratios of the core sizes (defined in Section 3). The algorithms are tested with random sparse third-order tensors as input. In general, the fit of the four algorithms follows the same pattern as in the first group of experiments (HOOI gives the best fit, then MP, SP, and HO-SVD), but we observe that SP is particularly sensitive to unbalanced ratios of the core sizes.

The third group explores the fit with varying ratios between the size of the input tensor and the size of the core tensor. For this group, we move from third-order tensors to fourth-order tensors. The algorithms are tested with random fourth-order tensors, with the input tensor size fixed while the core sizes vary. The fit of the algorithms follows the same pattern as in the previous two groups of experiments, in spite of the move to fourth-order tensors.

The final group measures the performance with a real (nonrandom) tensor that was generated for a task in computational linguistics. The fit follows the same pattern as in the previous three groups of experiments. Furthermore, the differences in fit are reflected in the performance on the given task. This experiment validates the use of random tensors in the previous three groups of experiments.

We conclude in Section 6. There are tradeoffs in time, space, and fit for the four algorithms, such that there is no absolute winner among the four algorithms. The choice will depend on the time, space, and fit requirements of the given application. If good fit is the primary concern, we recommend HOOI for smaller tensors that can fit in the available RAM, and MP for larger tensors.

2 Applications

A good survey of applications for tensor decompositions for data analysis is Acar and Yener (2007), which lists several applications, including electroencephalogram (EEG) data analysis, spectral analysis of chemical mixtures, computer vision, and social network analysis. Kolda (2006) also lists various applications, such as psychometrics, image analysis, graph analysis, and signal processing.

We believe that a natural place to look for applications for tensor decompositions is wherever SVD has proven useful. We have grown accustomed to thinking of data in terms of two-dimensional tables and matrices; in terms of what we can handle with SVD. However, real applications often have many more modes, which we have been ignoring.

In information retrieval (Deerwester et al., 1990), SVD is typically applied to a *term* $\times$ *document* matrix, where each row represents a word and each column represents a document in the collection. An element in the matrix is a weight that represents the importance of the given word in the given document. SVD smoothes the weights, so that a document d will have a nonzero weight for a word w if d is similar to other documents that contain the word w, even if d does not contain actually contain w. Thus a search for w will return the document d, thanks to the smoothing effect of SVD.

To extend the term-document matrix to a third-order tensor, it would be natural to add information such as author, date of publication, citations, and venue (e.g., the name of the conference or journal). For example, Dunlavy et al. (2006) used a tensor to combine information from abstracts, titles, keywords, authors, and citations. Chew et al. (2007) applied a tensor decomposition to a *term* $\times$ *document* $\times$ *language* tensor, for cross-language information retrieval. Sun et al. (2006) analyzed an *author* $\times$ *keyword* $\times$ *date* tensor.

In collaborative filtering (Billsus and Pazzani, 1998), SVD is usually applied to a *user* $\times$ *item* matrix, in which each row represents a person and each column represent an item, such as a movie or a book. An element in the matrix is a rating by the given user for the given item. Most of the elements in the ma-

trix are missing, because each user has only rated a few items. When a zero element represents a missing rating, SVD can be used to guess the missing ratings, based on the nonzero elements.

The user-item matrix could be extended to a third-order tensor by adding a variety of information, such as the words used to describe an item, the words used to describe the interests of a user, the price of an item, the geographical location of a user, and the age of a user. For example, Mahoney et al. (2006) and Xu et al. (2006) applied tensor decompositions to collaborative filtering.

In computational linguistics, SVD is often applied in semantic space models of word meaning. For example, Landauer and Dumais (1997) applied SVD to a *word* × *document* matrix, achieving human-level scores on multiple-choice synonym questions from the TOEFL test. Turney (2006) applied SVD to a *word-pair* × *pattern* matrix, reaching human-level scores on multiple-choice analogy questions from the SAT test.

In our recent work, we have begun exploring tensor decompositions for semantic space models. We are currently developing a *word* × *pattern* × *word* tensor that can used for both synonyms and analogies. The experiments in Section 5.4 evaluate the four tensor decomposition algorithms using this tensor to answer multiple-choice TOEFL questions.

In computational vision, SVD is often applied to image analysis (Brand, 2002). To work with the two-mode constraint of SVD, an image, which is naturally two-dimensional, is mapped to a vector. For example, in face recognition, SVD is applied to a *face* × *image-vector* matrix, in which each row is a vector that encodes an image of a person's face. Wang and Ahuja (2005) pointed out that this two-mode approach to image analysis is ignoring essential higher-mode structure in the data. The experiments in Wang and Ahuja (2005) demonstrate that higher-order tensor decompositions can be much more effective than SVD.

In summary, wherever SVD has been useful, we expect there are higher-order modes that have been ignored. With algorithms that can decompose large tensors, it is no longer necessary to ignore these modes.

3 Notation

This paper follows the notational conventions of Kolda (2006). Tensors of order three or higher are represented by bold script letters, $\boldsymbol{\mathcal{X}}$. Matrices (second-order tensors) are denoted by bold capital letters, $\mathbf{A}$. Vectors (first-order tensors) are denoted by bold lowercase letters, $\mathbf{b}$. Scalars (zero-order tensors) are represented by lowercase italic letters, i.

The i-th element in a vector $\mathbf{b}$ is indicated by b_i. The i-th row in a matrix $\mathbf{A}$ is denoted by $\mathbf{a}_{i:}$, the j-th column is given by $\mathbf{a}_{:j}$, and the element in row i and column j is represented by a_{ij}.

A third-order tensor $\boldsymbol{\mathcal{X}}$ has rows, columns, and *tubes*. The element in row i, column j, and tube k is represented by x_{ijk}. The row vector that contains x_{ijk} is denoted by $\mathbf{x}_{i:k}$, the column vector is $\mathbf{x}_{:jk}$, and the tube vector is $\mathbf{x}_{ij:}$. In general, the vectors in a tensor (e.g., the rows, columns, and tubes in a third-order tensor) are called *fibers*. There are no special names (beyond rows, columns, and tubes) for fibers in tensors of order four and higher.

A third-order tensor $\boldsymbol{\mathcal{X}}$ contains matrices, called *slices*. The horizontal, lateral, and frontal slices of $\boldsymbol{\mathcal{X}}$ are represented by $\mathbf{X}_{i::}$, $\mathbf{X}_{:j:}$, and $\mathbf{X}_{::k}$, respectively. The concept of slices also applies to tensors of order four and higher.

An index i ranges from 1 to I; that is, the upper bound on the range of an index is given by the uppercase form of the index letter. Thus the size of a tensor $\boldsymbol{\mathcal{X}}$ is denoted by uppercase scalars, $I_1 \times I_2 \times I_3$.

There are several kinds of tensor products, but we only need the n-mode product in this paper. The n-mode product of a tensor $\boldsymbol{\mathcal{X}}$ and a matrix $\mathbf{A}$ is written as $\boldsymbol{\mathcal{X}} \times_n \mathbf{A}$. Let $\boldsymbol{\mathcal{X}}$ be of size $I_1 \times I_2 \times I_3$ and let $\mathbf{A}$ be of size $J_1 \times J_2$. The n-mode product $\boldsymbol{\mathcal{X}} \times_n \mathbf{A}$ multiplies fibers in mode n of $\boldsymbol{\mathcal{X}}$ with row vectors in $\mathbf{A}$. Therefore n-mode multiplication requires that $I_n = J_2$. The result of $\boldsymbol{\mathcal{X}} \times_n \mathbf{A}$ is a tensor with the same order (same number of modes) as $\boldsymbol{\mathcal{X}}$, but with the size I_n replaced by J_1. For example, the result of $\boldsymbol{\mathcal{X}} \times_3 \mathbf{A}$ is of size $I_1 \times I_2 \times J_1$, assuming $I_3 = J_2$.

Let $\boldsymbol{\mathcal{X}}$ be an N-th order tensor of size $I_1 \times \ldots \times I_N$ and let $\mathbf{A}$ be a matrix of size $J \times I_n$. Suppose that $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{X}} \times_n \mathbf{A}$. Thus $\boldsymbol{\mathcal{Y}}$ is of size $I_1 \times \ldots \times I_{n-1} \times J \times I_{n+1} \times \ldots \times I_N$. The elements of $\boldsymbol{\mathcal{Y}}$ are defined as follows:

$$y_{i_1 \ldots i_{n-1} j i_{n+1} \ldots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 \ldots i_N} \, a_{j i_n} \qquad (1)$$

The transpose of a matrix $\mathbf{A}$ is written as $\mathbf{A}^{\mathsf{T}}$. We may think of the classical matrix product $\mathbf{AB}$ as a special case of n-mode product:

$$\mathbf{AB} = \mathbf{A} \times_2 \mathbf{B}^{\mathsf{T}} = \mathbf{B} \times_1 \mathbf{A} \qquad (2)$$

Fibers in mode two of $\mathbf{A}$ (row vectors) are multiplied with row vectors in $\mathbf{B}^\mathsf{T}$, which are column vectors (mode one) in $\mathbf{B}$.

A tensor $\mathcal{X}$ can be unfolded into a matrix, which is called *matricization*. The n-mode matricization of $\mathcal{X}$ is written $\mathbf{X}_{(n)}$, and is formed by taking the mode n fibers of $\mathcal{X}$ and making them column vectors in $\mathbf{X}_{(n)}$. Let $\mathcal{X}$ be a tensor of size $I_1 \times I_2 \times I_3$. The one-mode matricization $\mathbf{X}_{(1)}$ is of size $I_1 \times (I_2\ I_3)$:

$$\mathbf{X}_{(1)} = [\,\mathbf{x}_{:11}\ \mathbf{x}_{:21}\ \ldots\ \mathbf{x}_{:I_2 I_3}\,] \quad (3)$$
$$= [\,\mathbf{X}_{::1}\ \mathbf{X}_{::2}\ \ldots\ \mathbf{X}_{::I_3}\,] \quad (4)$$

Similarly, the two-mode matricization $\mathbf{X}_{(2)}$ is of size $I_2 \times (I_1\ I_3)$:

$$\mathbf{X}_{(2)} = [\,\mathbf{x}_{1:1}\ \mathbf{x}_{2:1}\ \ldots\ \mathbf{x}_{I_1:I_3}\,] \quad (5)$$
$$= [\,\mathbf{X}^\mathsf{T}_{::1}\ \mathbf{X}^\mathsf{T}_{::2}\ \ldots\ \mathbf{X}^\mathsf{T}_{::I_3}\,] \quad (6)$$

Note that $\mathcal{Y} = \mathcal{X} \times_n \mathbf{A}$ if and only if $\mathbf{Y}_{(n)} = \mathbf{A}\mathbf{X}_{(n)}$. Thus n-mode matricization relates the classical matrix product to the n-mode tensor product. In the special case of second-order tensors, $\mathbf{C}_{(1)} = \mathbf{C}$ and $\mathbf{C}_{(2)} = \mathbf{C}^\mathsf{T}$, hence $\mathbf{C} = \mathbf{B} \times_1 \mathbf{A}$ if and only if $\mathbf{C} = \mathbf{A}\mathbf{B}$. Likewise $\mathbf{C} = \mathbf{B} \times_2 \mathbf{A}$ if and only if $\mathbf{C}^\mathsf{T} = \mathbf{A}\mathbf{B}^\mathsf{T}$.

Let $\mathcal{G}$ be a tensor of size $J_1 \times \ldots \times J_N$. Let $\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}$ be matrices such that $\mathbf{A}^{(n)}$ is of size $I_n \times J_n$. The *Tucker operator* is defined as follows (Kolda, 2006):

$$[\![\mathcal{G}\,; \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!] \equiv \mathcal{G} \times_1 \mathbf{A}^{(1)} \times_2 \mathbf{A}^{(2)} \ldots \times_N \mathbf{A}^{(N)} \quad (7)$$

The resulting tensor is of size $I_1 \times \ldots \times I_N$.

Let $\mathcal{X}$ be a tensor of size $I_1 \times \ldots \times I_N$. The *Tucker decomposition* of $\mathcal{X}$ has the following form (Tucker, 1966; Kolda, 2006):

$$\mathcal{X} \approx [\![\mathcal{G}\,; \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!] \quad (8)$$

The tensor $\mathcal{G}$ is called the *core* of the decomposition. Let $\mathcal{G}$ be of size $J_1 \times \ldots \times J_N$. Each matrix $\mathbf{A}^{(n)}$ is of size $I_n \times J_n$.

The *n-rank* of a tensor $\mathcal{X}$ is the rank of the matrix $\mathbf{X}_{(n)}$. For a second-order tensor, the one-rank necessarily equals the two-rank, but this is not true for higher-order tensors. If J_n is equal to the n-rank of $\mathcal{X}$ for each n, then it is possible for the Tucker decomposition to exactly equal $\mathcal{X}$. In general, we want J_n less than the n-rank of $\mathcal{X}$ for each n, yielding a core $\mathcal{G}$ that has lower n-ranks than $\mathcal{X}$, analogous to a truncated (thin) SVD. In the special case of a second-order tensor, the Tucker decomposition $\mathbf{X} \approx [\![\mathbf{S}\,; \mathbf{U}, \mathbf{V}]\!]$ is equivalent to the thin SVD, $\mathbf{X} \approx \mathbf{U}\mathbf{S}\mathbf{V}^\mathsf{T}$.

Suppose we have a tensor $\mathcal{X}$ and its Tucker decomposition $\hat{\mathcal{X}} = [\![\mathcal{G}\,; \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!]$, such that $\mathcal{X} \approx \hat{\mathcal{X}}$. In the experiments in Section 5, we measure the *fit* of the decomposition $\hat{\mathcal{X}}$ to the original $\mathcal{X}$ as follows:

$$\text{fit}(\mathcal{X}, \hat{\mathcal{X}}) = 1 - \frac{\left\|\mathcal{X} - \hat{\mathcal{X}}\right\|_F}{\|\mathcal{X}\|_F} \quad (9)$$

The Frobenius norm of a tensor $\mathcal{X}$, $\|\mathcal{X}\|_F$, is the square root of the sum of the absolute squares of its elements. The fit is a normalized measure of the error in reconstructing $\mathcal{X}$ from its Tucker decomposition $\hat{\mathcal{X}}$. When $\mathcal{X} = \hat{\mathcal{X}}$, the fit is 1; otherwise, it is less than 1, and it may be negative when the fit is particularly poor.

The equivalence between the n-mode tensor product and the classical matrix product with n-mode matricization suggests that tensors might be merely a new notation; that there may be no advantage to using the Tucker decomposition with tensors instead of using SVD with unfolded (matricized) tensors. Perhaps the different layers (slices) of the tensor do not actually interact with each other in any interesting way. This criticism would be appropriate if the Tucker decomposition used only one mode, but the decomposition uses all N modes of $\mathcal{X}$. Because all modes are used, the layers of the tensor are thoroughly mixed together.

For example, suppose $\mathcal{X} \approx [\![\mathcal{G}\,; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$. Let $\mathbf{X}_{i::}$ be a slice of $\mathcal{X}$. There is no slice of $\mathcal{G}$, say $\mathbf{G}_{j::}$, such that we can reconstruct $\mathbf{X}_{i::}$ from $\mathbf{G}_{j::}$, using $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. We need all of $\mathcal{G}$ in order to reconstruct $\mathbf{X}_{i::}$.

All four of the algorithms that we examine in this paper perform the Tucker decomposition. One reason for our focus on the Tucker decomposition is that Bro and Andersson (1998) showed that the Tucker decomposition can be combined with other tensor decompositions, such as PARAFAC (Harshman, 1970; Carroll and Chang, 1970). In general, algorithms for the Tucker decomposition scale to large tensors better than most other tensor decomposition algorithms; therefore it is possible to improve the speed of other algorithms by first com-

pressing the tensor with the Tucker decomposition. The slower algorithm (such as PARAFAC) is then applied to the (relatively small) Tucker core, instead of the whole (large) input tensor (Bro and Andersson, 1998). Thus an algorithm that can perform the Tucker decomposition with large tensors makes it possible for other kinds of tensor decompositions to be applied to large tensors.

4 Algorithms

This section introduces the four tensor decomposition algorithms. All four algorithms take as input an arbitrary tensor $\mathcal{X}$ and a desired core size $J_1 \times \ldots \times J_N$ and generate as output a Tucker decomposition $\hat{\mathcal{X}} = [\![\mathcal{G}\,; \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!]$, in which the matrices $\mathbf{A}^{(n)}$ are orthonormal.

For HO-SVD (Higher-Order Singular Value Decomposition) and HOOI (Higher-Order Orthogonal Iteration), we show the algorithms specialized for third-order tensors and generalized for arbitrary tensors. For SP (Slice Projection) and MP (Multislice Projection), we present the algorithms for third-order and fourth-order tensors and leave the generalization for arbitrary tensors as an excercise for the reader. (There is a need for a better notation, to write the generalization of SP and MP to arbitrary tensors.)

4.1 Higher-Order SVD

Figure 1 presents the HO-SVD algorithm for third-order tensors. Figure 2 gives the generalization of HO-SVD for tensors of arbitrary order (De Lathauwer et al., 2000a; Kolda, 2006). In the following experiments, we used the implementation of HO-SVD in the MATLAB Tensor Toolbox (Bader and Kolda, 2007b). HO-SVD is not a distinct function in the Toolbox, but it is easily extracted from the Tucker Alternating Least Squares function, where it is a component.

HO-SVD does not attempt to optimize the fit, $\mathrm{fit}(\mathcal{X}, \hat{\mathcal{X}})$ (Kolda, 2006). That is, HO-SVD does not produce an optimal rank-$J_1, \ldots, J_N$ approximation to $\mathcal{X}$, because it optimizes for each mode separately, without considering interactions among the modes. However, we will see in Section 5 that HO-SVD often produces a reasonable approximation, and it is relatively fast. For more information about HO-SVD, see De Lathauwer et al. (2000a).

4.2 Higher-Order Orthogonal Iteration

Figure 3 presents the HOOI algorithm for third-order tensors. Figure 4 gives the generalization of HOOI for tensors of arbitrary order (De Lathauwer et al., 2000b; Kolda, 2006). HOOI is implemented in the MATLAB Tensor Toolbox (Bader and Kolda, 2007b), in the Tucker Alternating Least Squares function.

HOOI uses HO-SVD to initialize the matrices, before entering the main loop. The implementation in the MATLAB Tensor Toolbox gives the option of using a random initialization, but initialization with HO-SVD usually results in a better fit.

In the main loop, each matrix is optimized individually, while the other matrices are held fixed. This general method is called Alternating Least Squares (ALS). HOOI, SP, and MP all use ALS.

The main loop terminates when the change in fit drops below a threshold or when the number of iterations reaches a maximum, whichever comes first. To calculate the fit for each iteration, HOOI first calculates the core $\mathcal{G}$ using $[\![\mathcal{X}\,; \mathbf{A}^{(1)\mathsf{T}}, \ldots, \mathbf{A}^{(N)\mathsf{T}}]\!]$, and then calculates $\hat{\mathcal{X}}$ from $[\![\mathcal{G}\,; \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!]$. The change in fit is the fit of the Tucker decomposition after the t-th iteration of the main loop minus the fit from the previous iteration:

$$\Delta_{\mathrm{fit}}(t) = \mathrm{fit}(\mathcal{X}, \hat{\mathcal{X}}^{(t)}) - \mathrm{fit}(\mathcal{X}, \hat{\mathcal{X}}^{(t-1)}) \qquad (10)$$

In the experiments, we set the threshold for $\Delta_{\mathrm{fit}}(t)$ at 10^{-4} and we set the maximum number of iterations at 50. (These are the default values in the MATLAB Tensor Toolbox.) The main loop usually terminated after half a dozen iterations or fewer, with $\Delta_{\mathrm{fit}}(t)$ less than 10^{-4}.

As implemented in the MATLAB Tensor Toolbox, calculating the HO-SVD initialization, the intermediate tensor $\mathcal{Z}$, and the change in fit, $\Delta_{\mathrm{fit}}(t)$, requires bringing the entire input tensor $\mathcal{X}$ into RAM. Although sparse representations are used, this requirement limits the size of the tensors that we can process, as we see in Section 5.1. For more information about HOOI, see De Lathauwer et al. (2000b) and Kolda (2006).

4.3 Slice Projection

Figure 5 presents the SP algorithm for third-order tensors (Wang and Ahuja, 2005). Although Wang and Ahuja (2005) do not discuss tensors beyond the third-order, the SP algorithm generalizes to tensors of arbitrary order. For example, Figure 6 shows SP for fourth-order tensors.

Instead of using HO-SVD, Wang and Ahuja (2005) initialize SP randomly, to avoid bringing $\mathcal{X}$

in: Tensor $\mathcal{X}$ of size $I_1 \times I_2 \times I_3$.
in: Desired rank of core: $J_1 \times J_2 \times J_3$.

$\mathbf{A} \leftarrow J_1$ leading eigenvectors of $\mathbf{X}_{(1)}\mathbf{X}_{(1)}^\mathsf{T}$ – $\mathbf{X}_{(1)}$ is the unfolding of $\mathcal{X}$ on mode 1
$\mathbf{B} \leftarrow J_2$ leading eigenvectors of $\mathbf{X}_{(2)}\mathbf{X}_{(2)}^\mathsf{T}$
$\mathbf{C} \leftarrow J_3$ leading eigenvectors of $\mathbf{X}_{(3)}\mathbf{X}_{(3)}^\mathsf{T}$

$\mathcal{G} \leftarrow [\![\mathcal{X}\,; \mathbf{A}^\mathsf{T}, \mathbf{B}^\mathsf{T}, \mathbf{C}^\mathsf{T}]\!]$

out: $\mathcal{G}$ of size $J_1 \times J_2 \times J_3$ and orthonormal matrices $\mathbf{A}$ of size $I_1 \times J_1$, $\mathbf{B}$ of size $I_2 \times J_2$, and $\mathbf{C}$ of size $I_3 \times J_3$, such that $\mathcal{X} \approx [\![\mathcal{G}\,; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$.

Figure 1: Higher-Order Singular Value Decomposition for third-order tensors (De Lathauwer et al., 2000a).

in: Tensor $\mathcal{X}$ of size $I_1 \times I_2 \times \cdots \times I_N$.
in: Desired rank of core: $J_1 \times J_2 \times \cdots \times J_N$.

for $n = 1, \ldots, N$ **do**
    $\mathbf{A}^{(n)} \leftarrow J_n$ leading eigenvectors of $\mathbf{X}_{(n)}\mathbf{X}_{(n)}^\mathsf{T}$ – $\mathbf{X}_{(n)}$ is the unfolding of $\mathcal{X}$ on mode n
end for

$\mathcal{G} \leftarrow [\![\mathcal{X}\,; \mathbf{A}^{(1)\mathsf{T}}, \ldots, \mathbf{A}^{(N)\mathsf{T}}]\!]$

out: $\mathcal{G}$ of size $J_1 \times J_2 \times \cdots \times J_N$ and orthonormal matrices $\mathbf{A}^{(n)}$ of size $I_n \times J_n$ such that $\mathcal{X} \approx [\![\mathcal{G}\,; \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!]$.

Figure 2: Higher-Order Singular Value Decomposition for tensors of arbitrary order (De Lathauwer et al., 2000a).

in: Tensor $\mathcal{X}$ of size $I_1 \times I_2 \times I_3$.
in: Desired rank of core: $J_1 \times J_2 \times J_3$.

$\mathbf{B} \leftarrow J_2$ leading eigenvectors of $\mathbf{X}_{(2)}\mathbf{X}_{(2)}^{\mathsf{T}}$ – initialization via HO-SVD
$\mathbf{C} \leftarrow J_3$ leading eigenvectors of $\mathbf{X}_{(3)}\mathbf{X}_{(3)}^{\mathsf{T}}$

while not converged **do** – main loop
  $\mathcal{U} \leftarrow [\![\mathcal{X}\,; \mathbf{I}_1, \mathbf{B}^{\mathsf{T}}, \mathbf{C}^{\mathsf{T}}]\!]$
  $\mathbf{A} \leftarrow J_1$ leading eigenvectors of $\mathbf{U}_{(1)}\mathbf{U}_{(1)}^{\mathsf{T}}$
  $\mathcal{V} \leftarrow [\![\mathcal{X}\,; \mathbf{A}^{\mathsf{T}}, \mathbf{I}_2, \mathbf{C}^{\mathsf{T}}]\!]$
  $\mathbf{B} \leftarrow J_2$ leading eigenvectors of $\mathbf{V}_{(2)}\mathbf{V}_{(2)}^{\mathsf{T}}$
  $\mathcal{W} \leftarrow [\![\mathcal{X}\,; \mathbf{A}^{\mathsf{T}}, \mathbf{B}^{\mathsf{T}}, \mathbf{I}_3]\!]$
  $\mathbf{C} \leftarrow J_3$ leading eigenvectors of $\mathbf{W}_{(3)}\mathbf{W}_{(3)}^{\mathsf{T}}$
end while

$\mathcal{G} \leftarrow [\![\mathcal{X}\,; \mathbf{A}^{\mathsf{T}}, \mathbf{B}^{\mathsf{T}}, \mathbf{C}^{\mathsf{T}}]\!]$

out: $\mathcal{G}$ of size $J_1 \times J_2 \times J_3$ and orthonormal matrices $\mathbf{A}$ of size $I_1 \times J_1$, $\mathbf{B}$ of size $I_2 \times J_2$, and $\mathbf{C}$ of size $I_3 \times J_3$, such that $\mathcal{X} \approx [\![\mathcal{G}\,; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$.

Figure 3: Higher-Order Orthogonal Iteration for third-order tensors (De Lathauwer et al., 2000b; Kolda, 2006). Note that it is not necessary to initialize $\mathbf{A}$, since the **while** loop sets $\mathbf{A}$ using $\mathbf{B}$ and $\mathbf{C}$. $\mathbf{I}_i$ is the identity matrix of size $I_i \times I_i$.

in: Tensor $\mathcal{X}$ of size $I_1 \times I_2 \times \cdots \times I_N$.
in: Desired rank of core: $J_1 \times J_2 \times \cdots \times J_N$.

for $n = 2, \ldots, N$ **do** – initialization via HO-SVD
  $\mathbf{A}^{(n)} \leftarrow J_n$ leading eigenvectors of $\mathbf{X}_{(n)}\mathbf{X}_{(n)}^{\mathsf{T}}$
end for

while not converged **do** – main loop
  for $n = 1, \ldots, N$ **do**
    $\mathcal{Z} \leftarrow [\![\mathcal{X}\,; \mathbf{A}^{(1)\mathsf{T}}, \ldots, \mathbf{A}^{(n-1)\mathsf{T}}, \mathbf{I}_n, \mathbf{A}^{(n+1)\mathsf{T}}, \ldots, \mathbf{A}^{(N)\mathsf{T}}]\!]$
    $\mathbf{A}^{(n)} \leftarrow J_n$ leading eigenvectors of $\mathbf{Z}_{(n)}\mathbf{Z}_{(n)}^{\mathsf{T}}$
  end for
end while

$\mathcal{G} \leftarrow [\![\mathcal{X}\,; \mathbf{A}^{(1)\mathsf{T}}, \ldots, \mathbf{A}^{(N)\mathsf{T}}]\!]$

out: $\mathcal{G}$ of size $J_1 \times J_2 \times \cdots \times J_N$ and orthonormal matrices $\mathbf{A}^{(n)}$ of size $I_n \times J_n$ such that $\mathcal{X} \approx [\![\mathcal{G}\,; \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!]$.

Figure 4: Higher-Order Orthogonal Iteration for tensors of arbitrary order (De Lathauwer et al., 2000b; Kolda, 2006). $\mathbf{I}_n$ is the identity matrix of size $I_n \times I_n$.

in: Tensor $\mathcal{X}$ of size $I_1 \times I_2 \times I_3$.
in: Desired rank of core: $J_1 \times J_2 \times J_3$.

$\mathbf{C} \leftarrow$ random matrix of size $I_3 \times J_3$ – normalize columns to unit length

while not converged **do** – main loop

$$\mathbf{M}_{13} \leftarrow \sum_{i=1}^{I_2} \mathbf{X}_{:i:}\mathbf{C}\mathbf{C}^{\mathsf{T}}\mathbf{X}_{:i:}^{\mathsf{T}}$$ – slices on mode 2

$\mathbf{A} \leftarrow J_1$ leading eigenvectors of $\mathbf{M}_{13}\mathbf{M}_{13}^{\mathsf{T}}$

$$\mathbf{M}_{21} \leftarrow \sum_{i=1}^{I_3} \mathbf{X}_{::i}^{\mathsf{T}}\mathbf{A}\mathbf{A}^{\mathsf{T}}\mathbf{X}_{::i}$$ – slices on mode 3

$\mathbf{B} \leftarrow J_2$ leading eigenvectors of $\mathbf{M}_{21}\mathbf{M}_{21}^{\mathsf{T}}$

$$\mathbf{M}_{32} \leftarrow \sum_{i=1}^{I_1} \mathbf{X}_{i::}^{\mathsf{T}}\mathbf{B}\mathbf{B}^{\mathsf{T}}\mathbf{X}_{i::}$$ – slices on mode 1

$\mathbf{C} \leftarrow J_3$ leading eigenvectors of $\mathbf{M}_{32}\mathbf{M}_{32}^{\mathsf{T}}$

end while

$\mathcal{G} \leftarrow [\![\mathcal{X}\,; \mathbf{A}^{\mathsf{T}}, \mathbf{B}^{\mathsf{T}}, \mathbf{C}^{\mathsf{T}}]\!]$

out: $\mathcal{G}$ of size $J_1 \times J_2 \times J_3$ and orthonormal matrices $\mathbf{A}$ of size $I_1 \times J_1$, $\mathbf{B}$ of size $I_2 \times J_2$, and $\mathbf{C}$ of size $I_3 \times J_3$, such that $\mathcal{X} \approx [\![\mathcal{G}\,; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$.

Figure 5: Slice Projection for third-order tensors (Wang and Ahuja, 2005). Note that it is not necessary to initialize $\mathbf{A}$ and $\mathbf{B}$.

in: Tensor $\mathcal{X}$ of size $I_1 \times I_2 \times I_3 \times I_4$.
in: Desired rank of core: $J_1 \times J_2 \times J_3 \times J_4$.

$\mathbf{D} \leftarrow$ random matrix of size $I_4 \times J_4$ – normalize columns to unit length

while not converged **do** – main loop

$\mathbf{M}_{14} \leftarrow \sum_{i=1}^{I_2} \sum_{j=1}^{I_3} \mathbf{X}_{:ij:} \mathbf{D} \mathbf{D}^\mathsf{T} \mathbf{X}_{:ij:}^\mathsf{T}$ – slices on modes 2 and 3

$\mathbf{A} \leftarrow J_1$ leading eigenvectors of $\mathbf{M}_{14} \mathbf{M}_{14}^\mathsf{T}$

$\mathbf{M}_{21} \leftarrow \sum_{i=1}^{I_3} \sum_{j=1}^{I_4} \mathbf{X}_{::ij}^\mathsf{T} \mathbf{A} \mathbf{A}^\mathsf{T} \mathbf{X}_{::ij}$ – slices on modes 3 and 4

$\mathbf{B} \leftarrow J_2$ leading eigenvectors of $\mathbf{M}_{21} \mathbf{M}_{21}^\mathsf{T}$

$\mathbf{M}_{32} \leftarrow \sum_{i=1}^{I_1} \sum_{j=1}^{I_4} \mathbf{X}_{i::j}^\mathsf{T} \mathbf{B} \mathbf{B}^\mathsf{T} \mathbf{X}_{i::j}$ – slices on modes 1 and 4

$\mathbf{C} \leftarrow J_3$ leading eigenvectors of $\mathbf{M}_{32} \mathbf{M}_{32}^\mathsf{T}$

$\mathbf{M}_{43} \leftarrow \sum_{i=1}^{I_1} \sum_{j=1}^{I_2} \mathbf{X}_{ij::}^\mathsf{T} \mathbf{C} \mathbf{C}^\mathsf{T} \mathbf{X}_{ij::}$ – slices on modes 1 and 2

$\mathbf{D} \leftarrow J_4$ leading eigenvectors of $\mathbf{M}_{43} \mathbf{M}_{43}^\mathsf{T}$

end while

$\mathcal{G} \leftarrow [\![\mathcal{X} \,; \mathbf{A}^\mathsf{T}, \mathbf{B}^\mathsf{T}, \mathbf{C}^\mathsf{T}, \mathbf{D}^\mathsf{T}]\!]$

out: $\mathcal{G}$ of size $J_1 \times J_2 \times J_3 \times J_4$ and orthonormal matrices
$\mathbf{A}$ of size $I_1 \times J_1$, $\mathbf{B}$ of size $I_2 \times J_2$, $\mathbf{C}$ of size $I_3 \times J_3$,
and $\mathbf{D}$ of size $I_4 \times J_4$, such that $\mathcal{X} \approx [\![\mathcal{G} \,; \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}]\!]$.

Figure 6: Slice Projection for fourth-order tensors.

into RAM. In Figure 5, the matrix $\mathbf{C}$ is is filled with random numbers from the uniform distribution over $[0, 1]$ and then the columns are normalized.

Note that HO-SVD calculates each matrix from $\mathcal{X}$ alone, whereas HOOI calculates each matrix from $\mathcal{X}$ and all of the other matrices. SP lies between HO-SVD and HOOI, in that it calculates each matrix from $\mathcal{X}$ and one other matrix.

In the main loop, the input tensor $\mathcal{X}$ is processed one slice at a time, again to avoid bringing the whole tensor into RAM. Before entering the main loop, the first step is to calculate the slices and store each slice in a file. MP requires this same first step. The MATLAB source code for MP, given in the Appendix, shows how we calculate the slices of $\mathcal{X}$ without bringing all of $\mathcal{X}$ into RAM.

Our approach to constructing the slice files assumes that the input tensor is given in a sparse representation, in which each nonzero element of the tensor is described by one line in a file. The description consists of the indices that specify the location of the nonzero element, followed by the value of the nonzero element. For example, the element x_{ijk} of a third-order tensor $\mathcal{X}$ is described as $\langle i, j, k, x_{ijk} \rangle$. To calculate the n-mode slices, we first sort the input tensor file by mode n. For example, we generate two-mode slices by sorting on j, the second column of the input file. This puts all of the elements of an n-mode slice together consecutively in the file. After sorting on mode n, we can read the sorted file one slice at a time, writing each mode n slice to its own unique file.

To sort the input file, we use the Unix *sort* command. This command allows the user to specify the amount of RAM used by the sort buffer. In the following experiments, we arbitrarily set the buffer to 4 GiB, half the available RAM. (For Windows, the Unix *sort* command is included in Cygwin.)

The main loop terminates after a maximum number of iterations or when the core stops growing, whichever comes first. The growth of the core is measured as follows:

$$\Delta_{\mathcal{G}}(t) = 1 - \frac{\left\| \mathcal{G}^{(t-1)} \right\|_F}{\left\| \mathcal{G}^{(t)} \right\|_F} \qquad (11)$$

In this equation, $\mathcal{G}^{(t)}$ is the core after the t-th iteration. We set the threshold for $\Delta_{\mathcal{G}}(t)$ at 10^{-4} and we set the maximum number of iterations at 50. The main loop usually terminated after half a dozen iterations or fewer, with $\Delta_{\mathcal{G}}(t)$ less than 10^{-4}.

SP uses $\Delta_{\mathcal{G}}(t)$ as a proxy for $\Delta_{\text{fit}}(t)$, to avoid bringing $\mathcal{X}$ into RAM. With each iteration, as the estimates for the matrices improve, the core captures more of the variation in $\mathcal{X}$, resulting in growth of the core. It is not necessary to bring $\mathcal{X}$ into RAM in order to calculate $\mathcal{G}^{(t)}$; we can calculate $\mathcal{G}^{(t)}$ one slice at a time, as given in the Appendix.

For more information about SP, see Wang and Ahuja (2005). Wang et al. (2005) introduced another low RAM algorithm for higher-order tensors, based on blocks instead of slices.

4.4 Multislice Projection

Figure 7 presents the MP algorithm for third-order tensors. The MP algorithm generalizes to arbitrary order. Figure 8 shows MP for fourth-order tensors.

The basic structure of MP is taken from SP, but MP takes three ideas from HOOI: (1) use HO-SVD to initialize, instead of random initialization, (2) use fit to determine convergence, instead of using the growth of the core, (3) use all of the other matrices to calculate a given matrix, instead of using only one other matrix. Like SP, MP begins by calculating all of the slices of the input tensor and storing each slice in a file. See the Appendix for details.

We call the initialization *pseudo* HO-SVD initialization, because it is not exactly HO-SVD, as can be seen by comparing the initialization in Figure 3 with the initialization in Figure 7. Note that $\mathbf{X}_{(2)}$ in Figure 3 is of size $I_2 \times (I_1\ I_3)$, whereas $\mathbf{M}_2$ in Figure 7 is of size $I_2 \times I_2$, which is usually much smaller. HO-SVD brings the whole tensor into RAM, but pseudo HO-SVD processes one slice at a time.

The main loop terminates when the change in fit drops below a threshold or when the number of iterations reaches a maximum, whichever comes first. We calculate the fit one slice at a time, as given in the Appendix; it is not necessary to bring the whole input tensor into RAM in order to calculate the fit. We set the threshold for $\Delta_{\text{fit}}(t)$ at 10^{-4} and we set the maximum number of iterations at 50. The main loop usually terminated after half a dozen iterations or fewer, with $\Delta_{\text{fit}}(t)$ less than 10^{-4}.

The most significant difference between SP and MP is that MP uses all of the other matrices to calculate a given matrix. For example, $\mathbf{M}_{13}$ in Figure 5 is based on $\mathcal{X}$ and $\mathbf{C}$, whereas the corresponding $\mathbf{M}_1$ in Figure 7 is based on $\mathcal{X}$, $\mathbf{B}$, and $\mathbf{C}$. In this respect, MP is like HOOI, as we can see with the corresponding $\mathcal{U}$ in Figure 3. By slicing on two modes, instead of only one, we improve the fit of the tensor, as we shall see in the next section.

in: Tensor $\mathcal{X}$ of size $I_1 \times I_2 \times I_3$.
in: Desired rank of core: $J_1 \times J_2 \times J_3$.

$\mathbf{M}_2 \leftarrow \sum_{i=1}^{I_1} \mathbf{X}_{i::} \mathbf{X}_{i::}^\mathsf{T} + \sum_{i=1}^{I_3} \mathbf{X}_{::i}^\mathsf{T} \mathbf{X}_{::i}$ – pseudo HO-SVD initialization

$\mathbf{B} \leftarrow J_2$ leading eigenvectors of $\mathbf{M}_2 \mathbf{M}_2^\mathsf{T}$

$\mathbf{M}_3 \leftarrow \sum_{i=1}^{I_1} \mathbf{X}_{i::}^\mathsf{T} \mathbf{X}_{i::} + \sum_{i=1}^{I_2} \mathbf{X}_{:i:}^\mathsf{T} \mathbf{X}_{:i:}$

$\mathbf{C} \leftarrow J_3$ leading eigenvectors of $\mathbf{M}_3 \mathbf{M}_3^\mathsf{T}$

while not converged **do** – main loop

$\mathbf{M}_1 \leftarrow \sum_{i=1}^{I_3} \mathbf{X}_{::i} \mathbf{B}\mathbf{B}^\mathsf{T} \mathbf{X}_{::i}^\mathsf{T} + \sum_{i=1}^{I_2} \mathbf{X}_{:i:} \mathbf{C}\mathbf{C}^\mathsf{T} \mathbf{X}_{:i:}^\mathsf{T}$ – slices on modes 2 and 3

$\mathbf{A} \leftarrow J_1$ leading eigenvectors of $\mathbf{M}_1 \mathbf{M}_1^\mathsf{T}$

$\mathbf{M}_2 \leftarrow \sum_{i=1}^{I_3} \mathbf{X}_{::i}^\mathsf{T} \mathbf{A}\mathbf{A}^\mathsf{T} \mathbf{X}_{::i} + \sum_{i=1}^{I_1} \mathbf{X}_{i::} \mathbf{C}\mathbf{C}^\mathsf{T} \mathbf{X}_{i::}^\mathsf{T}$ – slices on modes 1 and 3

$\mathbf{B} \leftarrow J_2$ leading eigenvectors of $\mathbf{M}_2 \mathbf{M}_2^\mathsf{T}$

$\mathbf{M}_3 \leftarrow \sum_{i=1}^{I_2} \mathbf{X}_{:i:}^\mathsf{T} \mathbf{A}\mathbf{A}^\mathsf{T} \mathbf{X}_{:i:} + \sum_{i=1}^{I_1} \mathbf{X}_{i::}^\mathsf{T} \mathbf{B}\mathbf{B}^\mathsf{T} \mathbf{X}_{i::}$ – slices on modes 1 and 2

$\mathbf{C} \leftarrow J_3$ leading eigenvectors of $\mathbf{M}_3 \mathbf{M}_3^\mathsf{T}$

end while

$\mathcal{G} \leftarrow [\![\mathcal{X} \,;\, \mathbf{A}^\mathsf{T}, \mathbf{B}^\mathsf{T}, \mathbf{C}^\mathsf{T}]\!]$

out: $\mathcal{G}$ of size $J_1 \times J_2 \times J_3$ and orthonormal matrices $\mathbf{A}$ of size $I_1 \times J_1$, $\mathbf{B}$ of size $I_2 \times J_2$, and $\mathbf{C}$ of size $I_3 \times J_3$, such that $\mathcal{X} \approx [\![\mathcal{G} \,;\, \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$.

Figure 7: Multislice Projection for third-order tensors. MATLAB source code for this algorithm is provided in the Appendix.

in: Tensor $\mathcal{X}$ of size $I_1 \times I_2 \times I_3 \times I_4$.
in: Desired rank of core: $J_1 \times J_2 \times J_3 \times J_4$.

$\mathbf{M}_2 \leftarrow \sum_{i=1}^{I_3}\sum_{j=1}^{I_4} \mathbf{X}_{::ij}^{\mathsf{T}}\mathbf{X}_{::ij} + \sum_{i=1}^{I_1}\sum_{j=1}^{I_4} \mathbf{X}_{i::j}\mathbf{X}_{i::j}^{\mathsf{T}} + \sum_{i=1}^{I_1}\sum_{j=1}^{I_3} \mathbf{X}_{i:j:}\mathbf{X}_{i:j:}^{\mathsf{T}}$
$\mathbf{B} \leftarrow J_2$ leading eigenvectors of $\mathbf{M}_2\mathbf{M}_2^{\mathsf{T}}$
$\mathbf{M}_3 \leftarrow \sum_{i=1}^{I_2}\sum_{j=1}^{I_4} \mathbf{X}_{:i:j}^{\mathsf{T}}\mathbf{X}_{:i:j} + \sum_{i=1}^{I_1}\sum_{j=1}^{I_4} \mathbf{X}_{i::j}^{\mathsf{T}}\mathbf{X}_{i::j} + \sum_{i=1}^{I_1}\sum_{j=1}^{I_2} \mathbf{X}_{ij::}\mathbf{X}_{ij::}^{\mathsf{T}}$
$\mathbf{C} \leftarrow J_3$ leading eigenvectors of $\mathbf{M}_3\mathbf{M}_3^{\mathsf{T}}$
$\mathbf{M}_4 \leftarrow \sum_{i=1}^{I_2}\sum_{j=1}^{I_3} \mathbf{X}_{:ij:}^{\mathsf{T}}\mathbf{X}_{:ij:} + \sum_{i=1}^{I_1}\sum_{j=1}^{I_3} \mathbf{X}_{i:j:}^{\mathsf{T}}\mathbf{X}_{i:j:} + \sum_{i=1}^{I_1}\sum_{j=1}^{I_2} \mathbf{X}_{ij::}^{\mathsf{T}}\mathbf{X}_{ij::}$
$\mathbf{D} \leftarrow J_4$ leading eigenvectors of $\mathbf{M}_4\mathbf{M}_4^{\mathsf{T}}$

while not converged **do**
  $\mathbf{M}_1 \leftarrow \sum_{i=1}^{I_3}\sum_{j=1}^{I_4} \mathbf{X}_{::ij}\mathbf{B}\mathbf{B}^{\mathsf{T}}\mathbf{X}_{::ij}^{\mathsf{T}} + \sum_{i=1}^{I_2}\sum_{j=1}^{I_4} \mathbf{X}_{:i:j}\mathbf{C}\mathbf{C}^{\mathsf{T}}\mathbf{X}_{:i:j}^{\mathsf{T}} + \sum_{i=1}^{I_2}\sum_{j=1}^{I_3} \mathbf{X}_{:ij:}\mathbf{D}\mathbf{D}^{\mathsf{T}}\mathbf{X}_{:ij:}^{\mathsf{T}}$
  $\mathbf{A} \leftarrow J_1$ leading eigenvectors of $\mathbf{M}_1\mathbf{M}_1^{\mathsf{T}}$
  $\mathbf{M}_2 \leftarrow \sum_{i=1}^{I_3}\sum_{j=1}^{I_4} \mathbf{X}_{::ij}^{\mathsf{T}}\mathbf{A}\mathbf{A}^{\mathsf{T}}\mathbf{X}_{::ij} + \sum_{i=1}^{I_1}\sum_{j=1}^{I_4} \mathbf{X}_{i::j}\mathbf{C}\mathbf{C}^{\mathsf{T}}\mathbf{X}_{i::j}^{\mathsf{T}} + \sum_{i=1}^{I_1}\sum_{j=1}^{I_3} \mathbf{X}_{i:j:}\mathbf{D}\mathbf{D}^{\mathsf{T}}\mathbf{X}_{i:j:}^{\mathsf{T}}$
  $\mathbf{B} \leftarrow J_2$ leading eigenvectors of $\mathbf{M}_2\mathbf{M}_2^{\mathsf{T}}$
  $\mathbf{M}_3 \leftarrow \sum_{i=1}^{I_2}\sum_{j=1}^{I_4} \mathbf{X}_{:i:j}^{\mathsf{T}}\mathbf{A}\mathbf{A}^{\mathsf{T}}\mathbf{X}_{:i:j} + \sum_{i=1}^{I_1}\sum_{j=1}^{I_4} \mathbf{X}_{i::j}^{\mathsf{T}}\mathbf{B}\mathbf{B}^{\mathsf{T}}\mathbf{X}_{i::j} + \sum_{i=1}^{I_1}\sum_{j=1}^{I_2} \mathbf{X}_{ij::}\mathbf{D}\mathbf{D}^{\mathsf{T}}\mathbf{X}_{ij::}^{\mathsf{T}}$
  $\mathbf{C} \leftarrow J_3$ leading eigenvectors of $\mathbf{M}_3\mathbf{M}_3^{\mathsf{T}}$
  $\mathbf{M}_4 \leftarrow \sum_{i=1}^{I_2}\sum_{j=1}^{I_3} \mathbf{X}_{:ij:}^{\mathsf{T}}\mathbf{A}\mathbf{A}^{\mathsf{T}}\mathbf{X}_{:ij:} + \sum_{i=1}^{I_1}\sum_{j=1}^{I_3} \mathbf{X}_{i:j:}^{\mathsf{T}}\mathbf{B}\mathbf{B}^{\mathsf{T}}\mathbf{X}_{i:j:} + \sum_{i=1}^{I_1}\sum_{j=1}^{I_2} \mathbf{X}_{ij::}^{\mathsf{T}}\mathbf{C}\mathbf{C}^{\mathsf{T}}\mathbf{X}_{ij::}$
  $\mathbf{D} \leftarrow J_4$ leading eigenvectors of $\mathbf{M}_4\mathbf{M}_4^{\mathsf{T}}$
end while

$\mathcal{G} \leftarrow [\![\mathcal{X}\,; \mathbf{A}^{\mathsf{T}}, \mathbf{B}^{\mathsf{T}}, \mathbf{C}^{\mathsf{T}}, \mathbf{D}^{\mathsf{T}}]\!]$

out: $\mathcal{G}$ of size $J_1 \times J_2 \times J_3 \times J_4$ and orthonormal matrices
  $\mathbf{A}$ of size $I_1 \times J_1$, $\mathbf{B}$ of size $I_2 \times J_2$, $\mathbf{C}$ of size $I_3 \times J_3$,
  and $\mathbf{D}$ of size $I_4 \times J_4$, such that $\mathcal{X} \approx [\![\mathcal{G}\,; \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}]\!]$.

Figure 8: Multislice Projection for fourth-order tensors.

5 Experiments

This section presents the four groups of experiments. The hardware for these experiments was a computer with two dual-core AMD Opteron 64 processors, 8 GiB of RAM, and a 16 GiB swap file. The software was 64 bit Suse Linux 10.0, MATLAB R2007a, and MATLAB Tensor Toolbox Version 2.2 (Bader and Kolda, 2007b). The algorithms only used one of the four cores; we did not attempt to perform parallel processing, although SP and MP could be parallelized readily.

The input files are plain text files with one line for each nonzero value in the tensor. Each line consists of integers that give the location of the nonzero value in the tensor, followed by a single real number that gives the nonzero value itself. The input files are in text format, rather than binary format, in order to facilitate sorting the files.

The output files are binary MATLAB files, containing the tensor decompositions of the input files. The four algorithms generate tensor decompositions that are numerically different but structurally identical. That is, the numerical values are different, but, for a given input tensor, the four algorithms generate core tensors and matrices of the same size. Therefore the output file size does not depend on which algorithm was used.

5.1 Varying Tensor Sizes

The goal of this group of experiments was to evaluate the four algorithms on increasingly larger tensors, to discover how their performance scales with size. HO-SVD and HOOI assume that the input tensor fits in RAM, whereas SP and MP assume that the input tensor file must be read in blocks. We expected that HO-SVD and HOOI would eventually run out of RAM, but we could not predict precisely how the four algorithms would scale, in terms of fit, time, and space.

Table 1 summarizes the input test tensors for the first group of experiments. The test tensors are random sparse third-order tensors, varying in size from 250^3 to 2000^3. The number of nonzeros in the tensors varies from 1.6 million to 800 million. The nonzero values are random samples from a uniform distribution between zero and one.

Table 2 shows the results of the first group of experiments. HO-SVD and HOOI were only able to process the first four tensors, with sizes from 250^3 to 1000^3. The 1000^3 tensor required almost 16 GiB of RAM. The next tensor, 1250^3, required more RAM than was available (24 GiB; 8 GiB of actual RAM plus a 16 GiB swap file). On the other hand, SP and MP were able to process all eight tensors, up to 2000^3. Larger tensors are possible with SP and MP; the limiting factor becomes run time, rather than available RAM.

Figure 9 shows the fit of the four algorithms. HOOI has the best fit, followed by MP, then SP, and finally HO-SVD. The curves for HO-SVD and HOOI stop at 100 million nonzeros (the 1000^3 tensor), but it seems likely that the same trend would continue, if sufficient RAM were available to apply HO-SVD and HOOI to the larger tensors.

The fit is somewhat low, at about 4%, due to the difficulty of fitting a random tensor with a core size that is 0.1% of the size of the input tensor. However, we are interested in the relative ranking of the four algorithms, rather than the absolute fit. The results in Section 5.4 show that the ranking we see here, in Figure 9, is predictive of the relative performance on a real (nonrandom) task.

Figure 10 shows the RAM use of the algorithms. As we can see in Table 2, there are two components to the RAM use of SP and MP, the RAM used by *sort* and the RAM used by MATLAB. We arbitrarily set the sorting buffer to 4 GiB, which sets an upper bound on the RAM used by *sort*. A machine with less RAM could use a smaller sorting buffer. We have not experimented with the buffer size, but we expect that the buffer could be made much smaller, with only a slight increase in run time. The growth of the MATLAB component of RAM use of SP and MP is slow, especially in comparison to HO-SVD and HOOI.

Figure 11 gives the run time. For the smallest tensors, SP and MP take longer to run than HO-SVD and HOOI, because SP and MP make more use of files and less use of RAM. With a tensor size of 1000^3, both HO-SVD and HOOI use up the available hardware RAM (8 GiB) and need to use the virtual RAM (the 16 GiB swap file), which explains the sudden upward surge in Figure 11 at 100 million nonzeros. In general, the run time of SP and MP is competitive with HO-SVD and HOOI.

The results show that SP and MP can handle much larger tensors than HO-SVD and HOOI (800 million nonzeros versus 100 million nonzeros), with only a small penalty in run time for smaller tensors. However, HOOI yields a better fit than MP. If fit is important, we recommend HOOI for smaller tensors and MP for larger tensors. If speed is more important, we recommend HO-SVD for smaller tensors and SP for larger tensors.

Input tensor size ($I_1 \times I_2 \times I_3$)	Core size ($J_1 \times J_2 \times J_3$)	Density (% Nonzero)	Nonzeros (Millions)	Input file (GiB)	Output file (MiB)
$250 \times 250 \times 250$	$25 \times 25 \times 25$	10	1.6	0.03	0.3
$500 \times 500 \times 500$	$50 \times 50 \times 50$	10	12.5	0.24	1.5
$750 \times 750 \times 750$	$75 \times 75 \times 75$	10	42.2	0.81	4.3
$1000 \times 1000 \times 1000$	$100 \times 100 \times 100$	10	100.0	1.93	9.5
$1250 \times 1250 \times 1250$	$125 \times 125 \times 125$	10	195.3	3.88	17.7
$1500 \times 1500 \times 1500$	$150 \times 150 \times 150$	10	337.5	6.85	29.7
$1750 \times 1750 \times 1750$	$175 \times 175 \times 175$	10	535.9	11.03	46.0
$2000 \times 2000 \times 2000$	$200 \times 200 \times 200$	10	800.0	16.64	67.4

Table 1: Random sparse third-order tensors of varying size.

Algorithm	Tensor	Nonzeros (Millions)	Fit (%)	Run time (HH:MM:SS)	Matlab RAM (GiB)	Sort RAM (GiB)	Total RAM (GiB)
HO-SVD	250^3	1.6	3.890	00:00:24	0.21	0.00	0.21
HO-SVD	500^3	12.5	3.883	00:03:44	1.96	0.00	1.96
HO-SVD	750^3	42.2	3.880	00:14:42	6.61	0.00	6.61
HO-SVD	1000^3	100.0	3.880	01:10:13	15.66	0.00	15.66
HOOI	250^3	1.6	4.053	00:01:06	0.26	0.00	0.26
HOOI	500^3	12.5	3.982	00:09:52	1.98	0.00	1.98
HOOI	750^3	42.2	3.955	00:42:45	6.65	0.00	6.65
HOOI	1000^3	100.0	3.942	04:01:36	15.74	0.00	15.74
SP	250^3	1.6	3.934	00:01:21	0.01	1.41	1.42
SP	500^3	12.5	3.906	00:10:21	0.02	4.00	4.03
SP	750^3	42.2	3.896	00:34:39	0.06	4.00	4.06
SP	1000^3	100.0	3.893	01:43:20	0.11	4.00	4.12
SP	1250^3	195.3	3.890	03:16:32	0.21	4.00	4.22
SP	1500^3	337.5	3.888	06:01:47	0.33	4.00	4.33
SP	1750^3	535.9	3.886	09:58:36	0.54	4.00	4.54
SP	2000^3	800.0	3.885	15:35:21	0.78	4.00	4.79
MP	250^3	1.6	3.979	00:01:45	0.01	1.41	1.42
MP	500^3	12.5	3.930	00:13:55	0.03	4.00	4.03
MP	750^3	42.2	3.914	00:51:33	0.06	4.00	4.07
MP	1000^3	100.0	3.907	02:21:30	0.12	4.00	4.12
MP	1250^3	195.3	3.902	05:05:11	0.22	4.00	4.23
MP	1500^3	337.5	3.899	09:28:49	0.37	4.00	4.37
MP	1750^3	535.9	3.896	16:14:01	0.56	4.00	4.56
MP	2000^3	800.0	3.894	25:43:17	0.81	4.00	4.82

Table 2: Performance of the four algorithms with tensors of varying size.

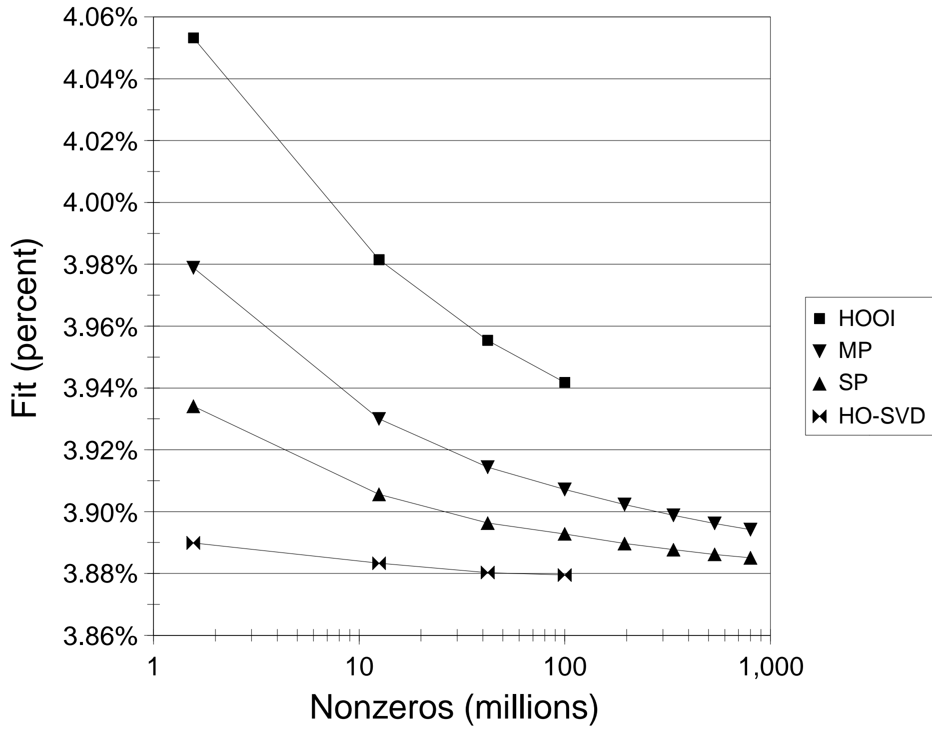

Figure 9: The fit of the four algorithms as a function of the number of nonzeros.

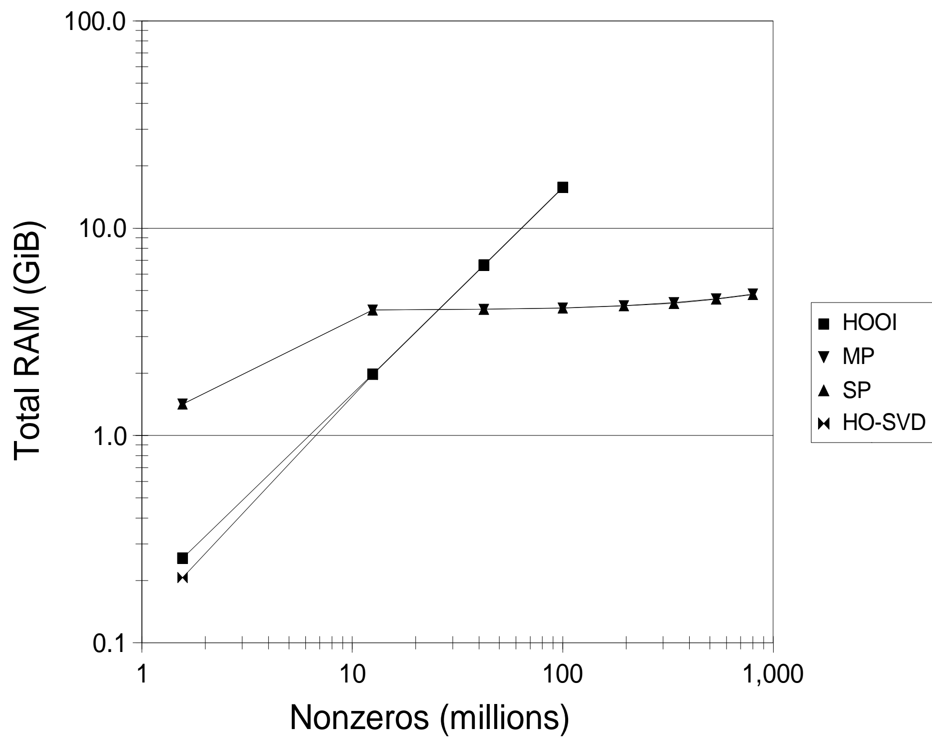

Figure 10: The RAM use of the four algorithms as a function of the number of nonzeros. Note that the size of the sorting buffer for SP and MP was arbitrarily set to 4 GiB.

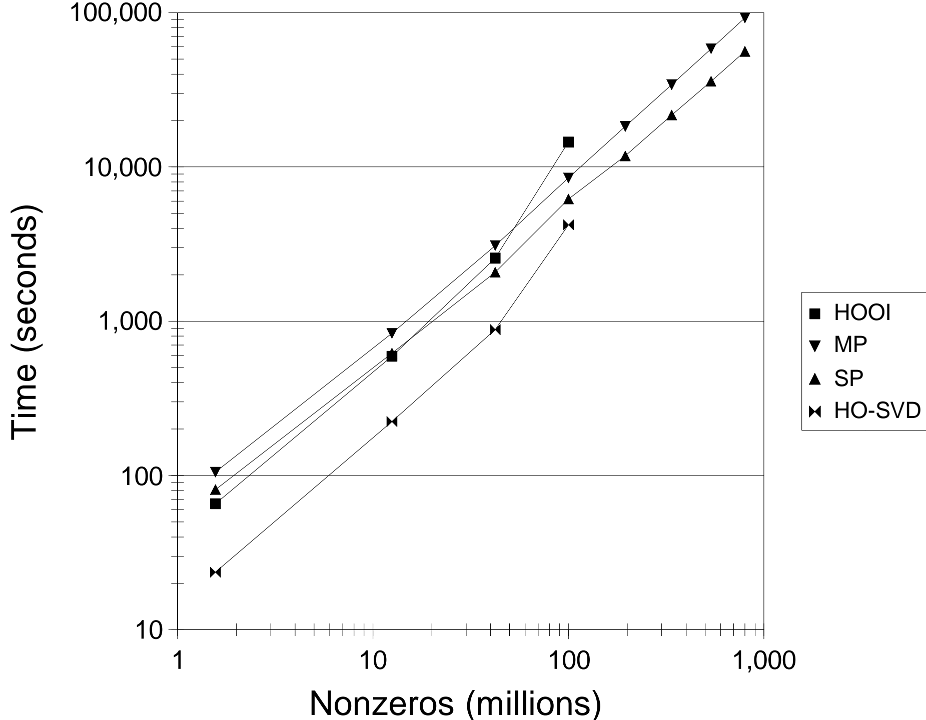

Figure 11: The run time of the four algorithms as a function of the number of nonzeros.

5.2 Varying Core Size Ratios

SP is somewhat different from the other three algorithms, in that it has a kind of asymmetry. Compare $\mathbf{M}_{13}$ in Figure 5 with $\mathbf{M}_1$ in Figure 7. We could have used $\mathbf{B}$ instead of $\mathbf{C}$, to calculate $\mathbf{A}$ in Figure 5, but we arbitrarily chose $\mathbf{C}$. We hypothesized that this asymmetry would make SP sensitive to variation in the ratios of the core sizes.

In this group of experiments, we vary the ratios between the sizes of the core in each mode, as listed in Table 3. The effect of the ratio on the performance is shown in Table 4. Figure 12 illustrates the effect of the ratio on the fit. It is clear from the figure that SP is asymmetrical, whereas HO-SVD, HOOI, and MP are symmetrical.

This asymmetry of SP might be viewed as a flaw, and thus a reason for preferring MP over SP, but it could also be seen as an advantage for SP. In the case where the ratio is 0.2, SP has a better fit than MP. This suggests that we might use SP instead of MP when the ratios between the sizes of the core in each mode are highly skewed; however, we must be careful to make sure that SP processes the matrices in the optimal order for the given core sizes.

Note that the relative ranking of the fit of the four algorithms is the same as in the previous group of experiments (best fit to worst: HOOI, MP, SP, HO-SVD), except in the case of extreme skew. Thus Figure 12 shows the robustness of the relative ranking.

5.3 Fourth-Order Tensors

This group of experiments demonstrates that the previous observations regarding the relative ranking of the fit also apply to fourth-order tensors. The experiments also investigate the effect of varying the size of the core, with a fixed input tensor size.

Table 5 lists the core sizes that we investigated. The effect of the core sizes on the performance is shown in Table 6. Figure 13 shows the impact of core size on fit.

The fit varies from about 4% with a core of 10^4 to about 44% with a core of 90^4. To make the differences among the algorithms clearer, we normalized the fit by using HO-SVD as a baseline. The fit relative to HO-SVD is defined as the percentage improvement in the fit of the given algorithm, compared to the fit of HO-SVD.

Figure 13 shows that the differences among the four algorithms are largest when the core is about 50^4; that is, the size of one mode of the core (50) is about half of the size of one mode of the input tensor

Input tensor size ($I_1 \times I_2 \times I_3$)	Core size ($J_1 \times J_2 \times J_3$)	Ratio ($J_1/J_2 = J_2/J_3$)	Density (%)	Nonzeros (Millions)	Input file (GiB)	Output file (MiB)
$500 \times 500 \times 500$	$250 \times 50 \times 10$	5.00	10	12.5	0.24	2.05
$500 \times 500 \times 500$	$125 \times 50 \times 20$	2.50	10	12.5	0.24	1.63
$500 \times 500 \times 500$	$83 \times 50 \times 30$	1.66	10	12.5	0.24	1.51
$500 \times 500 \times 500$	$63 \times 50 \times 40$	1.26	10	12.5	0.24	1.48
$500 \times 500 \times 500$	$50 \times 50 \times 50$	1.00	10	12.5	0.24	1.46
$500 \times 500 \times 500$	$40 \times 50 \times 63$	0.80	10	12.5	0.24	1.48
$500 \times 500 \times 500$	$30 \times 50 \times 83$	0.60	10	12.5	0.24	1.51
$500 \times 500 \times 500$	$20 \times 50 \times 125$	0.40	10	12.5	0.24	1.63
$500 \times 500 \times 500$	$10 \times 50 \times 250$	0.20	10	12.5	0.24	2.05

Table 3: Random sparse third-order tensors with varying ratios between the sizes of the core in each mode.

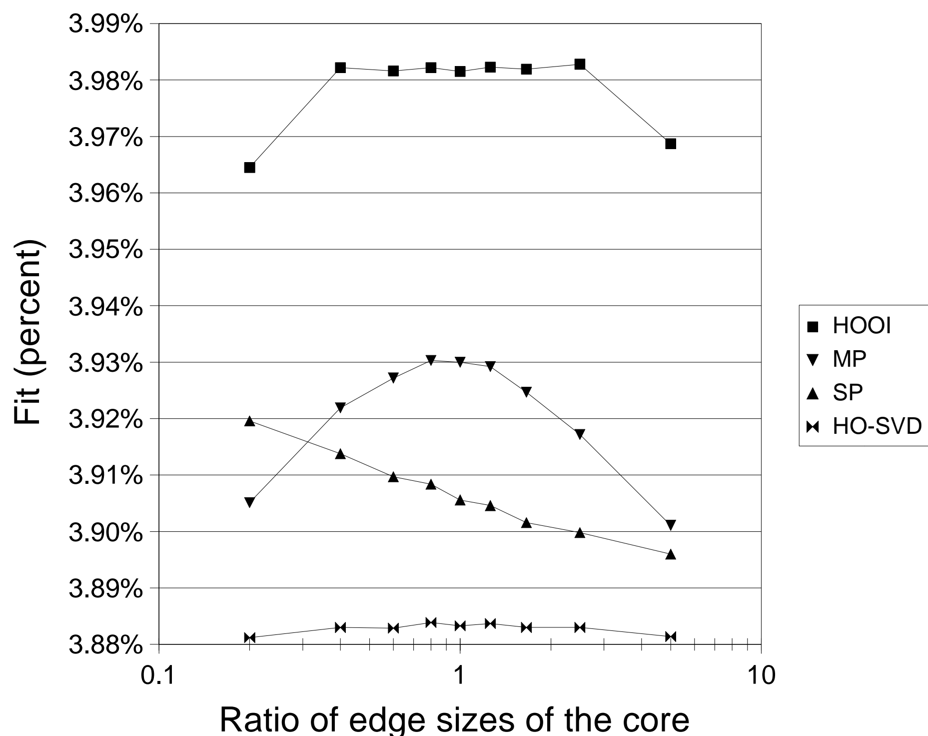

Figure 12: The fit of the four algorithms as a function of the ratios between the sizes of the core in each mode.

Algorithm	Ratio ($J_1/J_2 = J_2/J_3$)	Fit (%)	Run time (HH:MM:SS)	Matlab RAM (GiB)	Sort RAM (GiB)	Total RAM (GiB)
HO-SVD	5.00	3.881	00:06:54	2.71	0.00	2.71
HO-SVD	2.50	3.883	00:04:53	1.96	0.00	1.96
HO-SVD	1.66	3.883	00:04:15	1.78	0.00	1.78
HO-SVD	1.26	3.884	00:03:53	1.96	0.00	1.96
HO-SVD	1.00	3.883	00:03:48	1.96	0.00	1.96
HO-SVD	0.80	3.884	00:03:33	1.96	0.00	1.96
HO-SVD	0.60	3.883	00:03:24	1.96	0.00	1.96
HO-SVD	0.40	3.883	00:03:15	1.96	0.00	1.96
HO-SVD	0.20	3.881	00:03:06	1.96	0.00	1.96
HOOI	5.00	3.969	00:27:24	2.72	0.00	2.72
HOOI	2.50	3.983	00:16:23	2.02	0.00	2.02
HOOI	1.66	3.982	00:12:53	1.98	0.00	1.98
HOOI	1.26	3.982	00:11:06	1.98	0.00	1.98
HOOI	1.00	3.982	00:09:53	1.98	0.00	1.98
HOOI	0.80	3.982	00:09:02	1.98	0.00	1.98
HOOI	0.60	3.982	00:08:11	1.98	0.00	1.98
HOOI	0.40	3.982	00:07:26	1.99	0.00	1.99
HOOI	0.20	3.965	00:05:32	2.02	0.00	2.02
SP	5.00	3.896	00:11:18	0.02	4.00	4.02
SP	2.50	3.900	00:09:36	0.02	4.00	4.02
SP	1.66	3.902	00:09:30	0.02	4.00	4.02
SP	1.26	3.905	00:10:12	0.02	4.00	4.03
SP	1.00	3.906	00:10:13	0.02	4.00	4.03
SP	0.80	3.908	00:10:12	0.03	4.00	4.03
SP	0.60	3.910	00:10:23	0.03	4.00	4.03
SP	0.40	3.914	00:10:32	0.04	4.00	4.04
SP	0.20	3.920	00:12:43	0.06	4.00	4.07
MP	5.00	3.901	00:15:01	0.02	4.00	4.03
MP	2.50	3.917	00:14:05	0.02	4.00	4.02
MP	1.66	3.925	00:13:46	0.02	4.00	4.03
MP	1.26	3.929	00:13:47	0.02	4.00	4.03
MP	1.00	3.930	00:13:51	0.03	4.00	4.03
MP	0.80	3.930	00:13:45	0.03	4.00	4.03
MP	0.60	3.927	00:14:17	0.03	4.00	4.04
MP	0.40	3.922	00:14:37	0.04	4.00	4.04
MP	0.20	3.905	00:16:33	0.06	4.00	4.07

Table 4: Performance of the four algorithms with varying ratios between the sizes of the core in each mode.

Input tensor size ($I_1 \times I_2 \times I_3 \times I_4$)	Core size ($J_1 \times J_2 \times J_3 \times J_4$)	Density (%)	Nonzeros (Millions)	Input file (MiB)	Output file (MiB)
100^4	90^4	10	10	197.13	480.68
100^4	80^4	10	10	197.13	300.16
100^4	70^4	10	10	197.13	176.02
100^4	60^4	10	10	197.13	95.08
100^4	50^4	10	10	197.13	45.91
100^4	40^4	10	10	197.13	18.86
100^4	30^4	10	10	197.13	6.02
100^4	20^4	10	10	197.13	1.23
100^4	10^4	10	10	197.13	0.10

Table 5: Random sparse fourth-order tensors with varying core sizes.

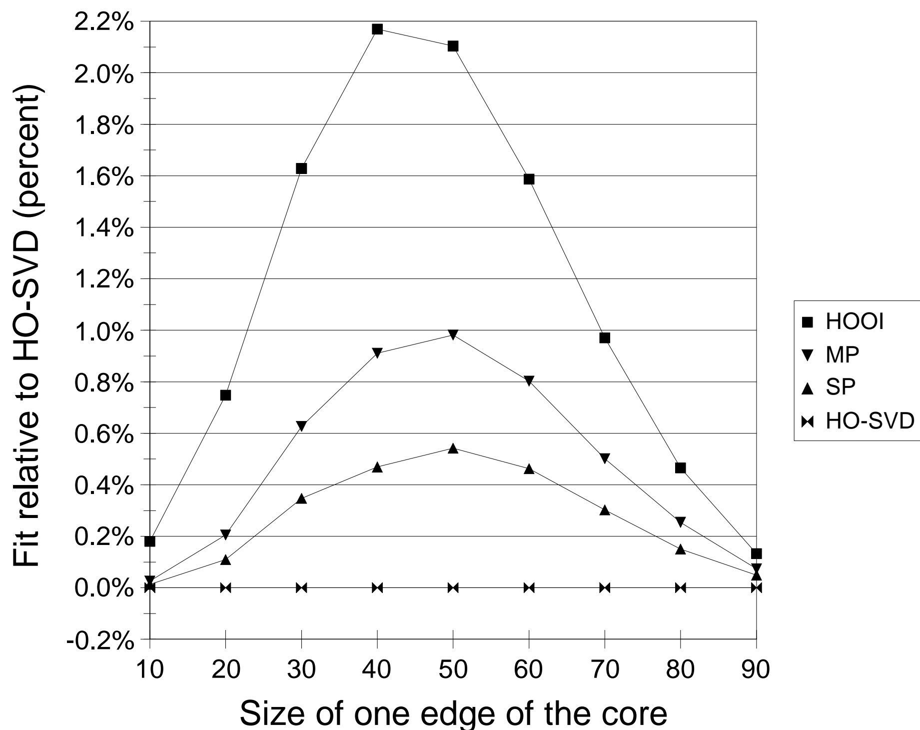

Figure 13: The fit of the four algorithms as a function of the core sizes, given fourth-order tensors.

Algorithm	Core Size	Fit (%)	Relative fit (%)	Run time (HH:MM:SS)	Matlab RAM (GiB)	Sort RAM (GiB)	Total RAM (GiB)
HO-SVD	90^4	44.007	0.00	00:05:35	3.56	0.00	3.56
HO-SVD	80^4	26.477	0.00	00:05:03	3.10	0.00	3.10
HO-SVD	70^4	16.449	0.00	00:04:20	3.02	0.00	3.02
HO-SVD	60^4	10.463	0.00	00:03:52	2.62	0.00	2.62
HO-SVD	50^4	6.988	0.00	00:03:27	2.07	0.00	2.07
HO-SVD	40^4	5.116	0.00	00:03:13	1.89	0.00	1.89
HO-SVD	30^4	4.232	0.00	00:02:57	1.80	0.00	1.80
HO-SVD	20^4	3.903	0.00	00:02:48	1.75	0.00	1.75
HO-SVD	10^4	3.827	0.00	00:02:34	1.80	0.00	1.80
HOOI	90^4	44.065	0.13	00:18:35	4.32	0.00	4.32
HOOI	80^4	26.600	0.47	00:21:16	3.75	0.00	3.75
HOOI	70^4	16.609	0.97	00:17:50	3.28	0.00	3.28
HOOI	60^4	10.629	1.59	00:15:00	2.88	0.00	2.88
HOOI	50^4	7.135	2.10	00:12:44	2.53	0.00	2.53
HOOI	40^4	5.227	2.17	00:10:19	2.24	0.00	2.24
HOOI	30^4	4.301	1.63	00:08:42	1.97	0.00	1.97
HOOI	20^4	3.933	0.75	00:05:25	1.81	0.00	1.81
HOOI	10^4	3.834	0.18	00:04:34	1.80	0.00	1.80
SP	90^4	44.029	0.05	01:45:07	2.19	4.00	6.19
SP	80^4	26.517	0.15	01:31:50	1.55	4.00	5.56
SP	70^4	16.499	0.30	01:17:53	1.06	4.00	5.07
SP	60^4	10.511	0.46	01:09:49	0.44	4.00	4.44
SP	50^4	7.026	0.54	01:04:21	0.38	4.00	4.38
SP	40^4	5.140	0.47	01:02:37	0.13	4.00	4.13
SP	30^4	4.247	0.35	01:01:09	0.07	4.00	4.08
SP	20^4	3.908	0.11	00:59:02	0.04	4.00	4.04
SP	10^4	3.828	0.01	00:57:56	0.01	4.00	4.02
MP	90^4	44.039	0.07	03:16:44	2.19	4.00	6.19
MP	80^4	26.544	0.25	02:31:07	1.55	4.00	5.56
MP	70^4	16.532	0.50	01:57:17	1.06	4.00	5.07
MP	60^4	10.547	0.80	01:36:45	0.69	4.00	4.70
MP	50^4	7.057	0.98	01:23:33	0.38	4.00	4.38
MP	40^4	5.163	0.91	01:14:23	0.17	4.00	4.18
MP	30^4	4.259	0.63	01:07:01	0.07	4.00	4.08
MP	20^4	3.911	0.20	01:04:29	0.04	4.00	4.04
MP	10^4	3.828	0.03	01:05:26	0.01	4.00	4.02

Table 6: Performance of the four algorithms with fourth-order tensors and varying core sizes. Relative fit is the percentage increase in fit relative to HO-SVD.

(100). When the core is very small or very large, compared to the input tensor, there is little difference in fit among the algorithms.

The fit follows the same trend here as in the previous two groups of experiments (best to worst: HOOI, MP, SP, HO-SVD), in spite of the switch from third-order tensors to fourth-order tensors. This further confirms the robustness of the results.

Table 6 shows that SP and MP are slow with fourth-order tensors, compared to HO-SVD and HOOI. This is a change from what we observered with third-order tensors, which did not yield such large differences in run time. This is because a fourth-order tensor has many more slices than a third-order tensor with the same number of elements, and each slice is smaller. There is a much larger overhead associated with opening and closing many small files, compared to a few large files. This could be ameliorated by storing several adjacent slices together in one file, instead of using a separate file for each slice.

Even with third-order tensors, grouping slices together in one file would improve the speed of SP and MP. Ideally, the user would specify the maximum RAM available and SP and MP would group as many slices together as would fit in the available RAM.

5.4 Performance with Real Data

So far, all our experiments have used random tensors. Our purpose with this last group of experiments is to show that the previous observations apply to nonrandom tensors. In particular, the differences in fit that we have seen so far are somewhat small. It seems possible that the differences might not matter in a real application of tensors. This group of experiments shows that the differences in fit result in differences in performance on a real task.

The task we examine here is answering multiple-choice synonym questions from the TOEFL test. This task was first investigated in Landauer and Dumais (1997). In ongoing work, we are exploring the application of third-order tensors to this task, combining ideas from Landauer and Dumais (1997) and Turney (2006).

Table 7 describes the input data and the output tensor decomposition. The first mode of the tensor consists of all of the 391 unique words that occur in the TOEFL questions. The second mode is a set of 849 words from Basic English, which is an artificial language that reduces English to a small, easily learned core vocabulary (Ogden, 1930). The third mode consists of 1020 patterns that join the words in the first two modes. These patterns were generated using the approach of Turney (2006). The value of an element in the tensor is derived from the frequency of the corresponding word pair and pattern in a large corpus.

A TOEFL question consists of a stem word (the target word) and four choice words. The task is to select the choice word that is most similar in meaning to the stem word. Our approach is to measure the similarity of two TOEFL words by the average similarity of their relations to the Basic English words.

Let $\mathcal{X}$ be our input tensor. Suppose we wish to measure the similarity of two TOEFL words. Let $\mathbf{X}_{i::}$ and $\mathbf{X}_{j::}$ be the slices of $\mathcal{X}$ that correspond to the two TOEFL words. Each slice gives the weights for all of the patterns that join the given TOEFL word to all of the Basic English words. Our measure of similarity between the TOEFL words is calculated by comparing the two slices.

Table 8 presents the performance of the four algorithms. We see that the fit follows the familiar pattern: HOOI has the best fit, then MP, next SP, and lastly HO-SVD. Note that MP and SP have similar fits. The final column of the table gives the TOEFL scores for the four algorithms. HOOI has the best TOEFL score, MP and SP have the same score, and HO-SVD has the lowest score. The bottom row of the table gives the TOEFL score for the raw input tensor, without the benefit of any smoothing from the Tucker decomposition. The results validate the previous experiments with random tensors and illustrate the value of the Tucker decomposition on a real task.

6 Conclusions

The Tucker decomposition has been with us since 1966, but it seems that it has only recently started to become popular. We believe that this is because only recently has computer hardware reached the point where large tensor decompositions are becoming feasible.

SVD started to attract interest in the field of information retrieval when it was applied to "problems of reasonable size (1000-2000 document abstracts; and 5000-7000 index terms)" (Deerwester et al., 1990). In collaborative filtering, SVD attracted interest when it achieved good results on the Netflix Prize, a dataset with a sparse matrix of 17,000 movies rated by 500,000 users. In realistic applications, size matters. The MATLAB Tensor Toolbox (Bader and Kolda, 2007a; Bader and Kolda, 2007b)

Input tensor size	$(I_1 \times I_2 \times I_3)$	$391 \times 849 \times 1020$
Core size	$(J_1 \times J_2 \times J_3)$	$250 \times 250 \times 250$
Input file	(MiB)	345
Output file	(MiB)	119
Density	(% Nonzero)	5.27
Nonzeros	(Millions)	18

Table 7: Description of the input data and the output decomposition.

Algorithm	Fit (%)	Relative fit (%)	Run time (HH:MM:SS)	Matlab RAM (GiB)	Sort RAM (GiB)	Total RAM (GiB)	TOEFL (%)
HO-SVD	21.716	0.00	00:10:28	5.29	0.00	5.29	80.00
HOOI	22.597	4.05	00:56:08	5.77	0.00	5.77	83.75
SP	22.321	2.78	00:30:02	0.33	4.00	4.33	81.25
MP	22.371	3.01	00:43:52	0.33	4.00	4.34	81.25
Raw tensor	-	-	-	-	-	-	67.50

Table 8: Performance of the four algorithms with actual data. Relative fit is the percentage increase in fit relative to HO-SVD.

has done much to make tensor decompositions more accessible and easier to experiment with, but, as we have seen here, RAM requirements become problematic with tensors larger than 1000^3.

The aim of this paper has been to empirically evaluate four tensor decompositions, to study their fit and their time and space requirements. Our primary concern was the ability of the algorithms to scale up to large tensors. The implementations of HO-SVD and HOOI, taken from the MATLAB Tensor Toolbox, assumed that the input tensor could fit in RAM, which limited them to tensors of size 1000^3. On the other hand, SP and MP were able to process tensors of size 2000^3, with eight times more elements.

The experiments in Section 5.4 suggest that the differences in fit among the four algorithms correspond to differences in performance on real tasks. It seems likely that good fit will be important for many applications; therefore, we recommend HOOI for those tensors that can fit in the available RAM, and MP for larger tensors.

Acknowledgements

Thanks to Brandyn Webb, Tamara Kolda, and Hongcheng Wang for helpful comments. Thanks to Tamara Kolda and Brett Bader for the MATLAB Tensor Toolbox.

References

E. Acar and B. Yener. 2007. Unsupervised multiway data analysis: A literature survey. Technical report, Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY. http://www.cs.rpi.edu/∼acare/Acar07_Multiway.pdf.

O. Alter, P.O. Brown, and D. Botstein. 2000. Singular value decomposition for genome-wide expression data processing and modeling. *Proceedings of the National Academy of Sciences*, 97(18):10101–10106.

B.W. Bader and T.G. Kolda. 2007a. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*.

B.W. Bader and T.G. Kolda. 2007b. MATLAB Tensor Toolbox version 2.2. http://csmr.ca.sandia.gov/∼tgkolda/TensorToolbox/.

D. Billsus and M.J. Pazzani. 1998. Learning collaborative information filters. *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 46–54.

M. Brand. 2002. Incremental singular value decomposition of uncertain data with missing values. *Proceedings of the 7th European Conference on Computer Vision*, pages 707–720.

R. Bro and C.A. Andersson. 1998. Improving the speed of multiway algorithms – Part II: Compression. *Chemometrics and Intelligent Laboratory Systems*, 42(1):105–113.

J.D. Carroll and J.J. Chang. 1970. Analysis of individual differences in multidimensional scaling via an n-way generalization of "Eckart-Young" decomposition. *Psychometrika*, 35(3):283–319.

P.A. Chew, B.W. Bader, T.G. Kolda, and A. Abdelali. 2007. Cross-language information retrieval using PARAFAC2. *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 143–152.

L. De Lathauwer, B. De Moor, and J. Vandewalle. 2000a. A multilinear singular value decomposition. *SIAM Journal on Matrix Analysis and Applications*, 21:1253–1278.

L. De Lathauwer, B. De Moor, and J. Vandewalle. 2000b. On the best rank-1 and rank-$(R_1, R_2, \ldots, R_N)$ approximation of higher-order tensors. *SIAM Journal on Matrix Analysis and Applications*, 21:1324–1342.

S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, and R. Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407.

D.M. Dunlavy, T.G. Kolda, and W.P. Kegelmeyer. 2006. Multilinear algebra for analyzing data with multiple linkages. Technical Report SAND2006-2079, Sandia National Laboratories, Livermore, CA. http://csmr.ca.sandia.gov/∼tgkolda/pubs/SAND2006-2079.pdf.

R.A. Harshman. 1970. Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multi-modal factor analysis. *UCLA Working Papers in Phonetics*, 16:1–84.

T.G. Kolda. 2006. Multilinear operators for higher-order decompositions. Technical Report SAND2006-2081, Sandia National Laboratories, Livermore, CA. http://csmr.ca.sandia.gov/∼tgkolda/pubs/SAND2006-2081.pdf.

T.K. Landauer and S.T. Dumais. 1997. A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological Review*, 104(2):211–240.

M.W. Mahoney, M. Maggioni, and P. Drineas. 2006. Tensor-CUR decompositions for tensor-based data. *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 327–336.

C.K. Ogden. 1930. *Basic English: A general introduction with rules and grammar*. Kegan Paul, Trench, Trubner and Co., London. http://ogden.basic-english.org/.

H. Schütze. 1998. Automatic word sense discrimination. *Computational Linguistics*, 24(1):97–123.

J. Sun, D. Tao, and C. Faloutsos. 2006. Beyond streams and graphs: Dynamic tensor analysis. *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 374–383.

L.R. Tucker. 1966. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311.

P.D. Turney. 2006. Similarity of semantic relations. *Computational Linguistics*, 32(3):379–416.

H. Wang and N. Ahuja. 2005. Rank-R approximation of tensors: Using image-as-matrix representation. *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, 2:346–353.

H. Wang, Q. Wu, L. Shi, Y. Yu, and N. Ahuja. 2005. Out-of-core tensor approximation of multi-dimensional matrices of visual data. *International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2005*, 24:527–535.

Y. Xu, L. Zhang, and W. Liu. 2006. Cubic analysis of social bookmarking for personalized recommendation. *Lecture Notes in Computer Science: Frontiers of WWW Research and Development – APWeb 2006*, 3841:733–738.

Appendix: MATLAB Source for Multislice Projection

```
--------------------------------------------------------------------------------

function fit = multislice(data_dir,sparse_file,tucker_file,I,J)
%MULTISLICE is a low RAM Tucker decomposition
%
%   Peter Turney
%   October 26, 2007
%
%   Copyright 2007, National Research Council of Canada
%
%   This program is free software: you can redistribute it and/or modify
%   it under the terms of the GNU General Public License as published by
%   the Free Software Foundation, either version 3 of the License, or
%   (at your option) any later version.
%
%   This program is distributed in the hope that it will be useful,
%   but WITHOUT ANY WARRANTY; without even the implied warranty of
%   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
%   GNU General Public License for more details.
%
%   You should have received a copy of the GNU General Public License
%   along with this program.  If not, see <http://www.gnu.org/licenses/>.
%
%%  set parameters
%
fprintf('MULTISLICE is running ...\n');
%
maxloops = 50;           % maximum number of iterations
eigopts.disp = 0;        % suppress messages from eigs()
minfitchange = 1e-4;     % minimum change in fit of tensor
%
%%  make slices of input data file
%
fprintf('   preparing slices\n');
%
mode1_dir = 'slice1';
mode2_dir = 'slice2';
mode3_dir = 'slice3';
%
slice(data_dir,sparse_file,mode1_dir,1,I);
slice(data_dir,sparse_file,mode2_dir,2,I);
slice(data_dir,sparse_file,mode3_dir,3,I);
%
%%  pseudo HO-SVD initialization
%
%   initialize B
%
M2 = zeros(I(2),I(2));
for i = 1:I(3)
    X3_slice = load_slice(data_dir,mode3_dir,i);
    M2 = M2 + (X3_slice' * X3_slice);
end
for i = 1:I(1)
    X1_slice = load_slice(data_dir,mode1_dir,i);
    M2 = M2 + (X1_slice * X1_slice');
end
[B,D] = eigs(M2*M2',J(2),'lm',eigopts);
%
%   initialize C
%
M3 = zeros(I(3),I(3));
for i = 1:I(1)
    X1_slice = load_slice(data_dir,mode1_dir,i);
    M3 = M3 + (X1_slice' * X1_slice);
end
for i = 1:I(2)
    X2_slice = load_slice(data_dir,mode2_dir,i);
    M3 = M3 + (X2_slice' * X2_slice);
end
```

```
[C,D] = eigs(M3*M3',J(3),'lm',eigopts);
%
%%  main loop
%
old_fit = 0;
%
fprintf('   entering main loop of MULTISLICE\n');
%
for loop_num = 1:maxloops
    %
    %   update A
    %
    M1 = zeros(I(1),I(1));
    for i = 1:I(2)
        X2_slice = load_slice(data_dir,mode2_dir,i);
        M1 = M1 + ((X2_slice * C) * (C' * X2_slice'));
    end
    for i = 1:I(3)
        X3_slice = load_slice(data_dir,mode3_dir,i);
        M1 = M1 + ((X3_slice * B) * (B' * X3_slice'));
    end
    [A,D] = eigs(M1*M1',J(1),'lm',eigopts);
    %
    %   update B
    %
    M2 = zeros(I(2),I(2));
    for i = 1:I(3)
        X3_slice = load_slice(data_dir,mode3_dir,i);
        M2 = M2 + ((X3_slice' * A) * (A' * X3_slice));
    end
    for i = 1:I(1)
        X1_slice = load_slice(data_dir,mode1_dir,i);
        M2 = M2 + ((X1_slice * C) * (C' * X1_slice'));
    end
    [B,D] = eigs(M2*M2',J(2),'lm',eigopts);
    %
    %   update C
    %
    M3 = zeros(I(3),I(3));
    for i = 1:I(1)
        X1_slice = load_slice(data_dir,mode1_dir,i);
        M3 = M3 + ((X1_slice' * B) * (B' * X1_slice));
    end
    for i = 1:I(2)
        X2_slice = load_slice(data_dir,mode2_dir,i);
        M3 = M3 + ((X2_slice' * A) * (A' * X2_slice));
    end
    [C,D] = eigs(M3*M3',J(3),'lm',eigopts);
    %
    %   build the core
    %
    G = zeros(I(1)*J(2)*J(3),1);
    G = reshape(G,[I(1) J(2) J(3)]);
    for i = 1:I(1)
        X1_slice = load_slice(data_dir,mode1_dir,i);
        G(i,:,:) = B' * X1_slice * C;
    end
    G = reshape(G,[I(1) (J(2)*J(3))]);
    G = A' * G;
    G = reshape(G,[J(1) J(2) J(3)]);
    %
    %   measure fit
    %
    normX = 0;
    sqerr = 0;
    for i = 1:I(1)
        X1_slice = load_slice(data_dir,mode1_dir,i);
        X1_approx = reshape(G,[J(1) (J(2)*J(3))]);
        X1_approx = A(i,:) * X1_approx;
        X1_approx = reshape(X1_approx,[J(2) J(3)]);
        X1_approx = B * X1_approx * C';
```

```
        sqerr = sqerr + norm(X1_slice-X1_approx,'fro')^2;
        normX = normX + norm(X1_slice,'fro')^2;
    end
    fit = 1 - sqrt(sqerr) / sqrt(normX);
    %
    fprintf('   loop %d: fit = %f\n', loop_num, fit);
    %
    %   stop if fit is not increasing fast enough
    %
    if ((fit - old_fit) < minfitchange)
        break;
    end
    %
    old_fit = fit;
    %
end
%
fprintf('   total loops = %d\n', loop_num);
%
%%  save tensor
%
output_file = [data_dir, '/', tucker_file];
save(output_file,'G','A','B','C');
%
fprintf('   tucker tensor is in %s\n',tucker_file);
%
fprintf('MULTISLICE is done\n');
%
```

```
function slice(data_dir,sparse_file,mode_slice_dir,mode,I)
%SLICE chops a tensor into slices along the given mode
%
%   Peter Turney
%   October 20, 2007
%
%   Copyright 2007, National Research Council of Canada
%
%%  initialize
%
%   set the secondary modes
%
if (mode == 1)
    r_mode = 2;
    c_mode = 3;
elseif (mode == 2)
    r_mode = 1;
    c_mode = 3;
else
    r_mode = 1;
    c_mode = 2;
end
%
%   get sizes
%
Ns = I(mode);        % number of slices
Nr = I(r_mode);      % number of rows in each slice
Nc = I(c_mode);      % number of columns in each slice
%
%%  sort the index
%
fprintf('SLICE is running ...\n');
%
%   file names
%
sub_dir     = [data_dir, '/', mode_slice_dir];
sorted_file = [sub_dir, '/', 'sorted.txt'];
%
%   make sure the directories exist
%
```

```
if (isdir(data_dir) == 0)
    mkdir(data_dir);
end
if (isdir(sub_dir) == 0)
    mkdir(sub_dir);
end
%
%   sort
%
sort_index(data_dir,sparse_file,mode_slice_dir,mode);
%
%%  count nonzeros in each slice
%
fprintf('   counting nonzeros in each slice for mode %d\n',mode);
%
%   vector for storing nonzero count
%
nonzeros = zeros(Ns,1);
%
%   read sorted file in blocks
%
%   - read in blocks because file may be too big to fit in RAM
%   - textscan will create one cell for each field
%   - each cell will contain a column vector of the values in
%     the given field
%   - the number of elements in each column vector is the number
%     of lines that were read
%
desired_lines = 100000;
actual_lines  = desired_lines;
%
sorted_file_id = fopen(sorted_file, 'r');
while (actual_lines > 0)
    block = textscan(sorted_file_id,'%d %d %d %*f',desired_lines);
    mode_subs = block{mode};
    actual_lines = size(mode_subs,1);
    for i = 1:actual_lines
        nonzeros(mode_subs(i)) = nonzeros(mode_subs(i)) + 1;
    end
end
fclose(sorted_file_id);
%
%%  make slices
%
fprintf('   saving slices for mode %d\n',mode);
%
sorted_file_id = fopen(sorted_file, 'r');
for i = 1:Ns
    slice_file = sprintf('%s/slice%d.mat', sub_dir, i);
    nonz = nonzeros(i);
    block = textscan(sorted_file_id,'%d %d %d %f',nonz);
    slice_rows = double(block{r_mode});
    slice_cols = double(block{c_mode});
    slice_vals = block{4};
    slice = sparse(slice_rows,slice_cols,slice_vals,Nr,Nc,nonz);
    save(slice_file,'slice');
end
fclose(sorted_file_id);
%
fprintf('SLICE is done\n');
%

--------------------------------------------------------------------------------

function sort_index(data_dir,sparse_file,mode_slice_dir,mode)
%SORT_INDEX sorts a sparse tensor index file along the given mode
%
%   Peter Turney
%   October 20, 2007
%
%   Copyright 2007, National Research Council of Canada
```

```
%
%% sort the index
%
fprintf('SORT_INDEX is running ...\n');
%
%   file names
%
input_file  = [data_dir, '/', sparse_file];
sub_dir     = [data_dir, '/', mode_slice_dir];
sorted_file = [sub_dir, '/', 'sorted.txt'];
%
%   call Unix 'sort' command
%
%   -n = numerical sorting
%   -k = key to sort on
%   -s = stable sorting
%   -S = memory for sorting buffer
%   -o = output file
%
%   - the 'sort' command is a standard part of Unix and Linux
%   - if you are running Windows, you can get 'sort' by
%     installing Cygwin
%   - the sort buffer is set here to 1 GiB; you can set it
%     to some other value, based on how much RAM you have
%
command = sprintf('sort -n -s -S 1G -k %d,%d -o %s %s', ...
    mode, mode, sorted_file, input_file);
%
fprintf('   calling Unix sort for mode %d\n', mode);
unix(command);
%
fprintf('SORT_INDEX is done\n');
%
```

```
function slice = load_slice(data_dir,mode_dir,i)
%LOAD_SLICE loads a sparse slice file
%
%   Peter Turney
%   October 20, 2007
%
%   Copyright 2007, National Research Council of Canada
%
%   file name
%
slice_file = sprintf('%s/%s/slice%d.mat', data_dir, mode_dir, i);
%
%   load the file
%
data = load(slice_file);
%
%   return the slice
%
slice = data.slice;
%
```

```
function test
%TEST illustrates how to use multislice.m
%
%   Peter Turney
%   October 26, 2007
%
%   Copyright 2007, National Research Council of Canada
%
%   test multislice.m
%
%   set random seed for repeatable experiments
%
```

```
rand('seed',5678);
%
%   set parameters
%
I = [100 110 120];      % input sparse tensor size
J = [10 11 12];         % desired core tensor size
density = 0.1;          % percent nonzero
%
data_dir    = 'test';         % directory for storing tensor
sparse_file = 'spten.txt';    % file for storing raw data tensor
tucker_file = 'tucker.mat';   % file for storing Tucker tensor
%
%   make a sparse random tensor and store it in a file
%
sparse_random_tensor(data_dir,sparse_file,I,density);
%
%   call multislice
%
tic;
fit = multislice(data_dir,sparse_file,tucker_file,I,J);
time = toc;
%
%   show results
%
fprintf('\n');
fprintf('Multislice:\n');
fprintf('I        = [%d %d %d]\n', I(1), I(2), I(3));
fprintf('J        = [%d %d %d]\n', J(1), J(2), J(3));
fprintf('density  = %f\n', density);
fprintf('fit      = %f\n', fit);
fprintf('time     = %.1f\n', time);
fprintf('\n');
%

--------------------------------------------------------------------------------

function sparse_random_tensor(data_dir,sparse_file,I,density)
%SPARSE_RANDOM_TENSOR makes a sparse uniformly distributed random tensor
%
%   Peter Turney
%   October 20, 2007
%
%   Copyright 2007, National Research Council of Canada
%
%   assume a third-order tensor is desired
%
%%  initialize
%
fprintf('SPARSE_RANDOM_TENSOR is running ...\n');
%
%   make sure the directory exists
%
if (isdir(data_dir) == 0)
    mkdir(data_dir);
end
%
file_name = [data_dir, '/', sparse_file];
%
fprintf('   generating tensor of size %d x %d x %d with density %f\n', ...
    I(1), I(2), I(3), density);
%
%%  main loop
%
file_id = fopen(file_name, 'w');
fprintf('   slice: ');
for i1 = 1:I(1)
    fprintf('%d ',i1);       % show progress
    if ((mod(i1,10) == 0) && (i1 ~= I(1)))
        fprintf('\n   ');    % time for new line
    end
    for i2 = 1:I(2)
```

```
        for i3 = 1:I(3)
            if (rand < density)
                fprintf(file_id,'%d %d %d %f\n',i1,i2,i3,rand);
            end
        end
    end
end
fprintf('\n');
fclose(file_id);
%
fprintf('SPARSE_RANDOM_TENSOR is done\n');
%
```