SANDIA REPORT

SAND2004-5187
Unlimited Release
Printed October 2004

MATLAB Tensor Classes for Fast Algorithm Prototyping

Brett W. Bader and Tamara G. Kolda

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Sandia National Laboratories

SAND2004-5187
Unlimited Release
Printed October 2004

MATLAB Tensor Classes for Fast Algorithm Prototyping

Brett W. Bader
Computational Sciences Department
Sandia National Laboratories
Albuquerque, NM 87185–0316
`bwbader@sandia.gov`

Tamara G. Kolda
Computational Sciences and Mathematics Research Department
Sandia National Laboratories
Livermore, CA 94551–9217
`tgkolda@sandia.gov`

Abstract

Tensors (also known as multidimensional arrays or N-way arrays) are used in a variety of applications ranging from chemometrics to psychometrics. We describe four MATLAB classes for tensor manipulations that can be used for fast algorithm prototyping. The `tensor` class extends the functionality of MATLAB's multidimensional arrays by supporting additional operations such as tensor multiplication. The `tensor_as_matrix` class supports the "matricization" of a tensor, i.e., the conversion of a tensor to a matrix (and vice versa), a commonly used operation in many algorithms. Two additional classes represent tensors stored in decomposed formats: `cp_tensor` and `tucker_tensor`. We describe all of these classes and then demonstrate their use by showing how to implement several tensor algorithms that have appeared in the literature.

Contents

Figures

MATLAB Tensor Classes for Fast Algorithm Prototyping

1 Introduction

A tensor is a multidimensional or N-way array of data; Figure 1 shows a 3-way array of size $I_1 \times I_2 \times I_3$. Tensors arise in many applications, including chemometrics [11], signal processing [3], and image processing [15]. In this paper, we describe four MATLAB classes for manipulating tensors: `tensor`, `tensor_as_matrix`, `cp_tensor`, and `tucker_tensor`.

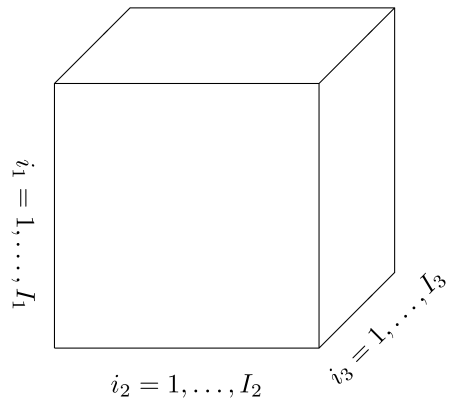

Figure 1. A 3-way array

MATLAB is a high-level computing environment that allows users to develop mathematical algorithms using familiar mathematical notation. In terms of higher-order tensors, MATLAB supports multidimensional arrays (MDAs). Allowed operations on MDAs include elementwise operations, permutation of indices, and most vector operations (like `sum` and `mean`) [13]. More complex operations, such as the multiplication of two MDAs, are not supported by MATLAB. This paper describes a `tensor` datatype that extends MATLAB's MDA functionality to support tensor multiplication and more through the use of MATLAB's class functionality [12].

Basic mathematical notation and operations for tensors, as well as the related MATLAB commands, are described in §2. Tensor multiplication receives its own section, §3, in which we describe both notation and how to multiply a tensor times a vector, a tensor times a matrix, and a tensor times another tensor. Conversion of a tensor to a matrix (and vice versa) via the `tensor_as_matrix` class is described in §4.

A tensor may be stored in factored form as a sum of rank-1 tensors. There are two commonly accepted factored forms. The first was developed independently under two names: the CANDECOMP model of Carroll and Chang [2] and the PARAFAC model of Harshman [7]. Following the notation in Kiers [9], we refer to this decomposition as the CP model. The second decomposition is the Tucker [14] model. Both models, as well as the corresponding MATLAB classes `cp_tensor` and `tucker_tensor`, are described in §5.

We note that these MATLAB classes serve a purely supporting role in the sense that these classes do not contain high-level algorithms—just the data types and their associated member functions. Thus, we view this work as complementary to those packages that provide algorithms for use with tensor data, the N-way toolbox for MATLAB by Andersson and Bro [1].

In general, we use the following notational conventions. Indices are denoted by lowercase letters and span the range from 1 to the uppercase letter of the index, e.g., $n = 1, 2, \ldots, N$. We denote vectors by lowercase boldface letters, e.g., $\mathbf{x}$; matrices by uppercase boldface, e.g., $\mathbf{U}$; and tensors by calligraphic letters, e.g., $\mathcal{A}$. Notation for tensor mathematics is still sometimes awkward. We have tried to be as standard as possible, relying on Harshman [8] and Kiers [9] for some guidance in this regard.

2 Basic Notation & MATLAB Commands for Tensors

Let $\mathcal{A}$ be a tensor of dimension $I_1 \times I_2 \times \cdots \times I_N$. The *order* of $\mathcal{A}$ is N. The nth *dimension* (or *mode* or *way*) of $\mathcal{A}$ is of size I_n.

A scalar is a zeroth-order tensor. An n-vector is a first-order tensor of size n. An $m \times n$ matrix is a second-order tensor of size $m \times n$. Of course, a single number could be a scalar, a 1-vector, a 1×1 matrix, etc. Similarly, an n-vector could be viewed as an $n \times 1$ matrix, or an $m \times n$ matrix could be viewed as a $m \times n \times 1$ tensor. It depends on the context, and our `tensor` class explicitly tracks the context, as described in §2.2.

We denote the index of a single element within a tensor by either subscripts or parentheses. Subscripts are generally used for indexing on matrices and vectors but can be confusing for the complex indexing that is sometimes required for tensors. In general, we use $\mathcal{A}(i_1, i_2, \ldots, i_N)$ rather than $\mathcal{A}_{i_1 i_2 \cdots i_N}$.

We use colon notation to denote the full range of a given index. The ith row of a matrix $\mathbf{A}$ is given by $\mathbf{A}(i, :)$, and the jth column is $\mathbf{A}(:, j)$. For higher-order tensors, the notation is extended in an obvious way, but the terminology is more complicated. Consider a 3rd-order tensor. In this case, specifying a single index yields a *slice* [9], which is a matrix in a specific orientation. So, $\mathcal{A}(i, :, :)$ yields the ith horizontal slice,

$\mathcal{A}(:, j, :)$ the jth lateral slice, and $\mathcal{A}(:, :, k)$ the kth frontal slice; see Figure 2.

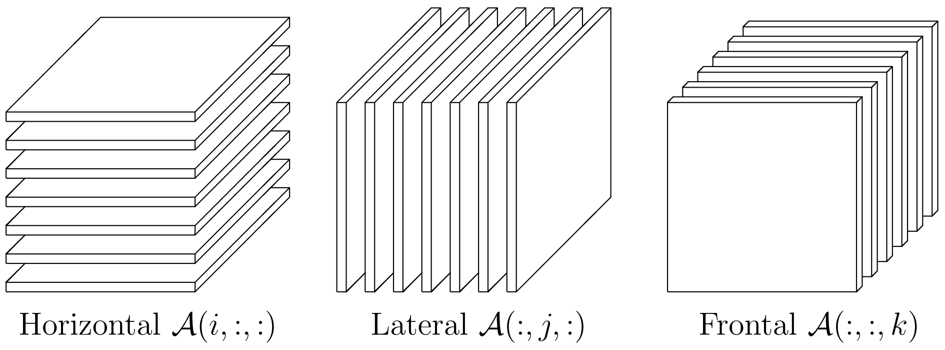
Horizontal $\mathcal{A}(i, :, :)$
Lateral $\mathcal{A}(:, j, :)$
Frontal $\mathcal{A}(:, :, k)$

Figure 2. Slices of a 3rd-order tensor.

On the other hand, $\mathcal{A}(:, j, k)$ yields a column vector, $\mathcal{A}(i, :, k)$ yields a row vector, and $\mathcal{A}(i, j, :)$ yields a so-called *tube vector* [10]; see Figure 3. Alternatively, these are called column fibers, row fibers, and depth fibers, respectively [9]. In general, a mode-n fiber is specified by fixing all dimensions except the nth.

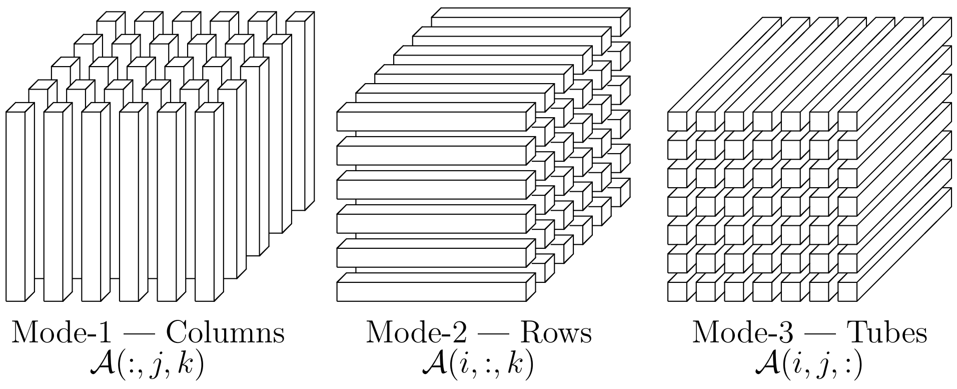

Figure 3. Fibers of a 3rd-order tensor.

2.1 Creating a `tensor` object

In MATLAB, a higher-order tensor can be stored as an MDA. We introduce the `tensor` class to extend the capabilities of the MDA datatype. An array or MDA can be converted to a tensor as follows, and Figure 4 shows an example of creating a tensor.

`T = tensor(A)` or `T = tensor(A,DIM)` converts an array (scalar, vector, matrix, or MDA) to a tensor. Here `A` is the object to be converted and `DIM` specifies the dimensions of the object.

`A = double(T)` converts a tensor to an array (scalar, vector, matrix, or MDA).

```
% Create a three-dimensional MDA
A = rand(3,4,2)

A(:,:,1) =
    0.2626    0.0211    0.8837    0.7377
    0.2021    0.0832    0.1891    0.3264
    0.7666    0.1450    0.4118    0.6331
A(:,:,2) =
    0.1501    0.0396    0.7307    0.4609
    0.2340    0.1489    0.6396    0.4528
    0.2955    0.4261    0.1215    0.1157

% Convert A to a tensor
T = tensor(A)

T is a tensor of size 3 x 4 x 2
T.data =
(:,:,1) =
    0.2626    0.0211    0.8837    0.7377
    0.2021    0.0832    0.1891    0.3264
    0.7666    0.1450    0.4118    0.6331
(:,:,2) =
    0.1501    0.0396    0.7307    0.4609
    0.2340    0.1489    0.6396    0.4528
    0.2955    0.4261    0.1215    0.1157
```

Figure 4. Example of creating a `tensor` object from a multidimensional array.

2.2 Tensors and size

Out of necessity, the `tensor` class handles sizes in a different way than the MATLAB arrays. Every MATLAB array has at least 2 dimensions; for example, a scalar is an object of size 1×1 and a column vector is an object of size $n \times 1$. On the other hand, MATLAB drops trailing singleton dimensions for any object of order greater than 2. Thus, a $4 \times 3 \times 1$ object has a reported size of 4×3. Our MATLAB `tensor` class explicitly stores trailing singleton dimensions; see Figure 5. Furthermore, the `tensor` class allows for tensors of order zero (for a scalar) or one (for a vector); see Figure 6. The function `order` returns the mathematical concept of order for a tensor, while the function `ndims` returns an algorithmic notion of the dimensions of a tensor, which is is useful for determining the number of subscripts capable of accessing all of the elements in the data structure (i.e., 1 for the case of a scalar or vector). Figure 5 also shows that the `whos` command does not report the correct sizes in the zero- or one-dimensional cases. The `tensor` constructor argument `DIM` must be specified whenever the order is intended to be zero or one or when there are trailing singleton dimensions.

2.3 Accessors

In MATLAB, indexing a tensor is the same as indexing a matrix:

`A(i1,i2,...,iN)` returns the $(i_1, i_2, \ldots, i_N)$ element of $\mathcal{A}$.

Recall that $\mathcal{A}(:,:,k)$ denotes the kth frontal slice. The MATLAB notation is straightforward:

`A(:,:,k)` returns the kth submatrix along the third dimension of the tensor $\mathcal{A}$.

Figure 7 shows examples of accessors for a tensor.

2.4 General functionality

In general, a `tensor` object will behave exactly as an MDA for all functions that are defined for an MDA; a list of these function is provided in Figure 8.

3 Tensor Multiplication

Notation for tensor multiplication is very complex. The issues have to do with specifying which dimensions are to be multiplied and how the dimensions of the result should be ordered. We approached this problem by developing notation that can be expressed easily by MATLAB. We describe three types of tensor multiplication: tensor times a matrix (§3.1), tensor times a vector (§3.2), and tensor times a tensor (§3.3).

```
% Create an MDA of size 4 x 3 x 1.
% Trailing singleton dimensions are ignored, so
% the number of reported dimensions is only 2.
A = rand([4 3 1]);

ndims(A)
ans =
     2

size(A)
ans =
     4     3

% Specifing the dimensions explicitly creates
% an order-3 tensor of size 4 x 3 x 1.
T = tensor(A,[4 3 1]);

ndims(T)
ans =
     3

size(T)
ans =
     4     3     1

% The ‘whos’ command reports the correct sizes.
whos

  Name      Size                    Bytes  Class

  A         4x3                        96  double array
  T         4x3x1                     368  tensor object

Grand total is 29 elements using 464 bytes
```

Figure 5. The `tensor` class explicitly keeps trailing singleton dimensions.

```
% A scalar can be stored as a tensor of order zero.
T0 = tensor(5,[]);

order(T0)
ans =
     0

ndims(T0)
ans =
     1

size(T0)
ans =
     []

% A vector can be stored as a tensor of order one.
T1 = tensor(rand(4,1),[4]);

order(T1)
ans =
     1

ndims(T1)
ans =
     1

size(T1)
ans =
     4

% The ‘whos’ command does not report the correct sizes!
whos

  Name      Size                   Bytes  Class

  T0        1x1                      256  tensor object
  T1        1x1                      288  tensor object

Grand total is 10 elements using 544 bytes
```

Figure 6. The `tensor` object can explicitly store zero- and one-dimensional objects.

```
% Create a random 2 x 2 x 2 tensor
A = tensor(rand(2,2,2))

A is a tensor of size 2 x 2 x 2
A.data =
(:,:,1) =
    0.4021    0.8332
    0.6531    0.3029
(:,:,2) =
    0.5953    0.3480
    0.4503    0.3982

% Access the (2,1,1) element
A(2,1,1)

ans =
    0.6531

% Reassign a 2 x 2 submatrix to be
% the 2 x 2 identity matrix
A(:,1,:) = eye(2)

A is a tensor of size 2 x 2 x 2
A.data =
(:,:,1) =
    1.0000    0.8332
         0    0.3029
(:,:,2) =
         0    0.3480
    1.0000    0.3982
```

Figure 7. Accessors and assignment for a `tensor` object work the same as they would for a multidimensional array.

- `A + B` or `plus(A,B)`
- `A - B` or `minus(A,B)`
- `-A` or `uminus(A)`
- `+A` or `uplus(A)`
- `A.*B` or `times(A,B)`
- `A./B` or `rdivide(A,B)`
- `A.\B` or `ldivide(A,B)`
- `A.^B` or `power(A,B)`

- `A < B` or `lt(A,B)`
- `A > B` or `gt(A,B)`
- `A <= B` or `le(A,B)`
- `A >= B` or `ge(A,B)`
- `A ~= B` or `ne(A,B)`
- `A == B` or `eq(A,B)`
- `A & B` or `and(A,B)`
- `A | B` or `or(A,B)`
- `~A` or `not(A)`

Figure 8. Functions that behave identically for tensors and multidimensional arrays.

3.1 Multiplying a tensor times a matrix

The first question we consider is how to multiply a tensor times a matrix. With matrix multiplication, the specification of which dimensions should be multiplied is straightforward—it is always the inner product of the rows of the first matrix with the columns of the second matrix. A transpose on an argument swaps the rows and columns. Because tensors may have an arbitrary number of dimensions, the situation is more complicated. In this case, we need to specify which dimension of the tensor is multiplied by the columns (or rows) of the given matrix.

The adopted solution is the n-mode product [5]. Let $\mathcal{A}$ be an $I_1 \times I_2 \times \cdots \times I_N$ tensor, and let $\mathbf{U}$ be an $J_n \times I_n$ matrix. Then the n-mode product of $\mathcal{A}$ and $\mathbf{U}$ is denoted by

$$\mathcal{A} \times_n \mathbf{U},$$

and defined (elementwise) as

$$(\mathcal{A} \times_n \mathbf{U})(i_1, \ldots, i_{n-1}, j_n, i_{n+1}, \ldots, i_N) = \sum_{i_n=1}^{I_n} \mathcal{A}(i_1, i_2, \ldots, i_N)\, \mathbf{B}(j_n, i_n).$$

The result is a tensor of size $I_1 \times \cdots \times I_{n-1} \times J_n \times I_{n+1} \times \cdots \times I_N$. Some authors call this operation the mode-n inner product and denote it as $\mathcal{A} \bullet_n \mathbf{U}$ (see, e.g., Comon [4]).

To understand n-mode multiplication in terms of matrices (i.e., order-2 tensors), suppose $\mathbf{A}$ is $m \times n$, $\mathbf{U}$ is $m \times k$, and $\mathbf{V}$ is $n \times k$. It follows that

$$\mathbf{A} \times_1 \mathbf{U}^T = \mathbf{U}^T\mathbf{A} \quad \text{and} \quad \mathbf{A} \times_2 \mathbf{V}^T = \mathbf{AV}.$$

Further, the matrix SVD can be written as

$$\mathbf{A} = \mathbf{U\Sigma V}^T = \mathbf{\Sigma} \times_1 \mathbf{U} \times_2 \mathbf{V}.$$

The following MATLAB commands can be used to calculate n-mode products.

`B = ttm(A,U,n)` calculates "tensor times matrix" in mode-n, i.e., $\mathcal{B} = \mathcal{A} \times_n \mathbf{U}$.

`B = ttm(A,{U,V},[m,n])` calculates two sequential n-mode products in the specified modes, i.e., $\mathcal{B} = \mathcal{A} \times_m \mathbf{U} \times_n \mathbf{V}$.

The n-mode product satisfies the following property [5]. Let $\mathcal{A}$ be a tensor of size $I_1 \times I_2 \times \cdots \times I_N$. If $\mathbf{U} \in \mathbb{R}^{J_m \times I_m}$ and $\mathbf{V} \in \mathbb{R}^{J_n \times I_n}$, then

$$\mathcal{A} \times_m \mathbf{U} \times_n \mathbf{V} = \mathcal{A} \times_n \mathbf{V} \times_m \mathbf{U}. \tag{1}$$

Figure 9 shows an example that demonstrates this property, and Figure 10 revisits the same example but calculates the products using cell arrays.

It is often desirable to calculate the product of a tensor and a sequence of matrices. Let $\mathcal{A}$ be an $I_1 \times I_2 \times \cdots \times I_N$ tensor, and let $\mathbf{U}^{(n)}$ denote a $J_n \times I_n$ matrix for $n = 1, \ldots, N$. Then the sequence of products

$$\mathcal{B} = \mathcal{A} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \cdots \times_N \mathbf{U}^{(N)} \tag{2}$$

is of size $J_1 \times J_2 \times \cdots \times J_N$. We propose new, alternative notation for this operation that is consistent with the MATLAB notation for cell arrays:

$$\mathcal{B} = \mathcal{A} \times \{\mathbf{U}\}.$$

This mathematical notation will prove useful in presenting some algorithms, as shown in §6.

The following equivalent MATLAB commands can be used to calculate n-mode products with a sequence of matrices.

`B = ttm(A,{U1,U2,...,UN}, [1:N])` calculates $\mathcal{B} = \mathcal{A} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \cdots \times_n \mathbf{U}^{(N)}$. Here `Un` is a MATLAB matrix representing $\mathbf{U}^{(n)}$.

`B = ttm(A,U)` calculates $\mathcal{B} = \mathcal{A} \times \{\mathbf{U}\}$. Here `U = {U1,U2,...,UN}` is a MATLAB cell array and `Un` is as described above.

```
A = tensor(rand(4,3,2));
U = rand(2,4);
V = rand(3,2);

% Computing A x_1 U x_3 V
B = ttm(A,U,1);
C = ttm(B,V,3)

C is a tensor of size 2 x 3 x 3
C.data =
(:,:,1) =
    1.9727    2.0380    2.6528
    1.6460    1.8647    2.4649
(:,:,2) =
    1.9051    2.0078    2.5385
    1.5881    1.8406    2.3523
(:,:,3) =
    0.3289    0.3437    0.4400
    0.2743    0.3148    0.4082

% Computing A x_3 V x_1 U
B = ttm(A,V,3);
C = ttm(B,U,1)

C is a tensor of size 2 x 3 x 3
C.data =
(:,:,1) =
    1.9727    2.0380    2.6528
    1.6460    1.8647    2.4649
(:,:,2) =
    1.9051    2.0078    2.5385
    1.5881    1.8406    2.3523
(:,:,3) =
    0.3289    0.3437    0.4400
    0.2743    0.3148    0.4082
```

Figure 9. Calculating n-mode products.

```
% Repeat the calculation from the previous figure,
% but use a cell array
W{1} = U;
W{2} = V;
C = ttm(A,W,[1 3])

C is a tensor of size 2 x 3 x 3
C.data =
(:,:,1) =
    1.9727    2.0380    2.6528
    1.6460    1.8647    2.4649
(:,:,2) =
    1.9051    2.0078    2.5385
    1.5881    1.8406    2.3523
(:,:,3) =
    0.3289    0.3437    0.4400
    0.2743    0.3148    0.4082
```

Figure 10. An alternate approach to calculating n-mode products.

Another frequently used operation is multiplying by *all but one* of a sequence of matrices:

$$\mathcal{B} = \mathcal{A} \times_1 \mathbf{U}^{(1)} \cdots \times_{n-1} \mathbf{U}^{(n-1)} \times_{n+1} \mathbf{U}^{(n+1)} \cdots \times_N \mathbf{U}^{(N)}.$$

We propose new, alternative notation for this operation:

$$\mathcal{B} = \mathcal{A} \times_{-n} \{\mathbf{U}\}.$$

This notation will prove useful in presenting some algorithms in §6.

The following MATLAB commands can be used to calculate n-mode products with all but one of a sequence of matrices.

`B = ttm(A,U,-n)` calculates $\mathcal{B} = \mathcal{A} \times_{-n} \{\mathbf{U}\}$. Here `U = {U1,U2,...,UN}` is a MATLAB cell array; the nth cell is simply ignored in the computation.

Note that `B = ttm(A,{U1,...,U4,U6,...,U9},[1:4,6:9])` is equivalent to `B = ttm(A,U,-5)` where `U={U1,...,U9}`; both calculate $\mathcal{B} = \mathcal{A} \times_{-5} \{\mathbf{U}\}$.

3.2 Multiplying a tensor times a vector

In our opinion, one source of confusion in n-mode multiplication is what to do to the singleton dimension in mode n that is introduced when multiplying a tensor times a vector. If the singleton dimension is dropped (as is sometimes desired), then the commutativity of the multiplies (1) outlined in the previous section no longer holds because the order of the intermediate result changes and $\times_n$ or $\times_m$ applies to the wrong mode.

Although one can usually determine the correct order of the result via the context of the equation, it is impossible to do this automatically in MATLAB in any robust way. Thus, we propose an alternate name and notation in the case when the newly introduced singleton dimension indeed should be dropped.

Let $\mathcal{A}$ be an $I_1 \times I_2 \times \cdots \times I_N$ tensor, and let $\mathbf{b}$ be an I_n-vector. We propose that the *contracted* n-mode product, which drops the nth singleton dimension, be denoted by

$$\mathcal{A} \bar{\times}_n \mathbf{b}.$$

The result is of size $I_1 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N$. Note that the order of the result is $N - 1$, one less than the original tensor. The entries are computed as:

$$(\mathcal{A} \bar{\times}_n \mathbf{u})(i_1, \ldots, i_{n-1}, i_{n+1}, \ldots, i_N) = \sum_{i_n=1}^{I_n} \mathcal{A}(i_1, i_2, \ldots, i_N)\ \mathbf{u}(i_n).$$

The following MATLAB command computes the contracted n-mode product.

B = `ttv(A,u,n)` calculates "tensor times vector" in mode-n, i.e., $B = \mathcal{A} \bar{\times}_n \mathbf{u}$.

Observe that $\mathcal{A} \bar{\times}_n \mathbf{u}$ and $\mathcal{A} \times_n \mathbf{u}^T$ produce identical results except for the order and shape of the results; that is, $\mathcal{A} \bar{\times}_n \mathbf{u}$ is of size $I_1 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N$, whereas $\mathcal{A} \times_n \mathbf{u}^T$ is of size $I_1 \times \cdots \times I_{n-1} \times 1 \times I_{n+1} \times \cdots \times I_N$. See Figure 11 for an example.

Because the contracted n-mode product drops the nth singleton dimension, it is no longer true that multiplication is commutative; i.e.,

$$(\mathcal{A} \bar{\times}_m \mathbf{u})\ \bar{\times}_n \mathbf{v} \neq (\mathcal{A} \bar{\times}_n \mathbf{v})\ \bar{\times}_m \mathbf{u}.$$

However, a different statement about commutativity may be made. If we assume $m < n$, then

$$(\mathcal{A} \bar{\times}_m \mathbf{u})\ \bar{\times}_{n-1} \mathbf{v} = (\mathcal{A} \bar{\times}_n \mathbf{v})\ \bar{\times}_m \mathbf{u}.$$

For the sake of clarity in a sequence of contracted products, we assume that the order reduction happens after all products have been computed. This assumption obviates

```
A = tensor(rand(3,4,2));
u = rand(3,1);

% Computing "tensor times vector"
C = ttv(A,u,1)

C is a tensor of size 4 x 2
C.data =
    0.5058    0.9385
    0.3319    0.4829
    0.1857    0.5141
    0.6210    0.8288

% Computing "tensor times matrix" yields same result,
% but of size 1 x 4 x 2.
B = ttm(A,u',1)

B is a tensor of size 1 x 4 x 2
B.data =
(:,:,1) =
    0.5058    0.3319    0.1857    0.6210
(:,:,2) =
    0.9385    0.4829    0.5141    0.8288
```

Figure 11. Comparison of $\mathcal{A} \times_n \mathbf{u}^T$ and $\mathcal{A} \mathbin{\bar{\times}_n} \mathbf{u}$.

the need to explicitly place parentheses in an expression and appropriately decrement any n-mode product indices. In other words,

$$\mathcal{A} \bar{\times}_m \mathbf{u} \bar{\times}_n \mathbf{v} \equiv \begin{cases} (\mathcal{A} \bar{\times}_m \mathbf{u}) \bar{\times}_n \mathbf{v} & : \quad m > n, \\ (\mathcal{A} \bar{\times}_m \mathbf{u}) \bar{\times}_{n-1} \mathbf{v} & : \quad m < n. \end{cases} \tag{3}$$

As before with matrices, it is often useful to calculate the product of a tensor and a sequence of vectors; e.g.,

$$\mathcal{B} = \mathcal{A} \bar{\times}_1 \mathbf{u}^{(1)} \bar{\times}_2 \mathbf{u}^{(2)} \cdots \bar{\times}_N \mathbf{u}^{(N)}$$

or

$$\mathcal{B} = \mathcal{A} \bar{\times}_1 \mathbf{u}^{(1)} \cdots \bar{\times}_{n-1} \mathbf{u}^{(n-1)} \bar{\times}_{n+1} \mathbf{u}^{(n+1)} \cdots \bar{\times}_N \mathbf{u}^{(N)}.$$

We propose the following alternative notation for these two cases:

$$\mathcal{B} = \mathcal{A} \bar{\times} \{\mathbf{u}\}$$

and

$$\mathcal{B} = \mathcal{A} \bar{\times}_{-n} \{\mathbf{u}\},$$

respectively.

In practice, one must be careful when calculating a sequence of contracted products to perform the multiplications starting with the highest mode and proceed sequentially to the lowest mode. The following MATLAB commands automatically sort the modes in the correct order.

`B = ttv(A, u1,u3, [1,3])` computes $B = \mathcal{A} \bar{\times}_1 \mathbf{u}^{(1)} \bar{\times}_3 \mathbf{u}^{(3)}$ where `u1` and `u3` correspond to vectors $\mathbf{u}^{(1)}$ and $\mathbf{u}^{(3)}$, respectively.

`B = ttv(A,u)` computes $B = \mathcal{A} \bar{\times} \{\mathbf{u}\}$ where `u` is a cell array whose nth entry is the vector $\mathbf{u}^{(n)}$.

`B = ttv(A,u,-n)` computes $B = \mathcal{A} \bar{\times}_{-n} \{\mathbf{u}\}$.

Note that the result of the second expression is a scalar, and the result of the third expression is a vector of size I_n; see Figure 12.

3.3 Multiplying a tensor times another tensor

The last category of tensor multiplication to consider is the product of two tensors. We consider three general scenarios for tensor-tensor multiplication: outer product, contracted product, and inner product.

```
A = tensor(rand(3,4,2));
U = {rand(3,1), rand(4,1), rand(2,1)};

% A tensor times a sequence of vectors in every
% dimension produces a scalar.
ttv(A,U)

ans is a tensor of order 0 (i.e., a scalar)
ans.data =
    1.1812

% A tensor times a sequence of vectors in every
% dimension *except one* produces a vector.
ttv(A,U,-2)

ans is a tensor of size 4
ans.data =
    0.5794
    0.9246
    0.8470
    0.3318
```

Figure 12. Tensor times a sequence of vectors.

The outer product of two tensors is defined as follows. Let $\mathcal{A}$ be of size $I_1 \times \cdots \times I_M$, and let $\mathcal{B}$ be of size $J_1 \times \cdots \times J_N$. The outer product $\mathcal{A} \circ \mathcal{B}$ is of size $I_1 \times \cdots \times I_M \times J_1 \times \cdots \times J_N$ and is given by

$$\mathcal{A} \circ \mathcal{B}(i_1, \ldots i_M, j_1, \ldots, j_N) = \mathcal{A}(i_1, \ldots, i_M)\mathcal{B}(j_1, \ldots, j_N).$$

In MATLAB, the command is as follows, and Figure 13 shows an example of computing the outer product of two tensors.

C= `ttt(A,B)` computes "tensor times tensor"; i.e., the outer product $\mathcal{C} = \mathcal{A} \circ \mathcal{B}$.

```
A = tensor(reshape([1:12],[3,4]));
B = tensor([1;2],2);

ttt(A,B)

ans is a tensor of size 3 x 4 x 2
ans.data =
(:,:,1) =
     1     4     7    10
     2     5     8    11
     3     6     9    12
(:,:,2) =
     2     8    14    20
     4    10    16    22
     6    12    18    24
```

Figure 13. Tensor times tensor: outer product.

The contracted product of two tensors is similar to tensor-vector multiplication and to tensor-matrix multiplication discussed above. However, a key distinction in this case is the specification of modes for an inner product computation and the ordering of the remaining modes in the product.

Specifically, let $\mathcal{A}$ be of size $I_1 \times \cdots \times I_M \times J_1 \times \cdots \times J_N$ and $\mathcal{B}$ be of size $I_1 \times \cdots \times I_M \times K_1 \times \cdots \times K_P$. We can multiply both tensors along the first M modes, and the result is a tensor of size $J_1 \times \cdots \times J_N \times K_1 \times \cdots \times K_P$, given by

$$\langle \mathcal{A}, \mathcal{B} \rangle_{\{1,\ldots,M;1,\ldots,M\}} (j_1, \ldots j_N, k_1, \ldots, k_P) = \sum_{i_1=1}^{I_1} \cdots \sum_{i_M=1}^{I_M} \mathcal{A}(i_1, \ldots, i_M, j_1, \ldots, j_N)\, \mathcal{B}(i_1, \ldots, i_M, k_1, \ldots, k_P).$$

With this notation, the modes to be multiplied are specified in the subscripts that follow the angle brackets. The remaining modes are ordered such that those from $\mathcal{A}$ come before $\mathcal{B}$, which is different from the tensor-matrix product case considered above where the leftover matrix dimension of $\mathbf{B}$ replaces I_n rather than moved to the end. In MATLAB, the command is as follows.

C = `ttt(A,B,[1:M],[1:M])` computes $\mathcal{C} = \langle \mathcal{A}, \mathcal{B} \rangle_{\{1,\ldots,M;1,\ldots,M\}}$.

The arguments specifying the modes of $\mathcal{A}$ and the modes of $\mathcal{B}$ for contraction need not be consecutive, as shown in the previous example. However, the sizes of the corresponding dimensions must be equal. That is, if we call `ttt(A,B,ADIMS,BDIMS)` then `size(A,ADIMS)` and `size(B,BDIMS)` must be identical. See Figure 14 for an example.

The inner product of two tensors requires that both tensors have equal dimensions. Assuming both are of size $I_1 \times I_2 \times \cdots \times I_N$, their inner product is given by

$$\langle \mathcal{A}, \mathcal{B} \rangle = \sum_{i_1=1}^{I_1} \sum_{i_2=1}^{I_2} \cdots \sum_{i_N=1}^{I_N} \mathcal{A}(i_1, i_2, \ldots, i_N)\, \mathcal{B}(i_1, i_2, \ldots, i_N).$$

In MATLAB this is accomplished via the following command; see Figure 15 for an example.

`ttt(A,B,[1:N])` calculates $\langle \mathcal{A}, \mathcal{B} \rangle$; the result is a scalar.

Using this definition of inner product, then the Frobenius norm of a tensor is given by

$$\|\mathcal{A}\|_F^2 = \langle \mathcal{A}, \mathcal{A} \rangle = \sum_{i_1=1}^{I_1} \sum_{i_2=1}^{I_2} \cdots \sum_{i_N=1}^{I_N} \mathcal{A}(i_1, i_2, \ldots, i_N)^2.$$

In MATLAB the norm can be calculated as follows.

`norm(A)` calculates $\|\mathcal{A}\|_F$, the Frobenius norm of a tensor.

```
A = tensor(rand(3,4,2))

A is a tensor of size 3 x 4 x 2
A.data =
(:,:,1) =
    0.9218    0.4057    0.4103    0.3529
    0.7382    0.9355    0.8936    0.8132
    0.1763    0.9169    0.0579    0.0099
(:,:,2) =
    0.1389    0.6038    0.0153    0.9318
    0.2028    0.2722    0.7468    0.4660
    0.1987    0.1988    0.4451    0.4186

B = tensor(rand(4,3,2))

B is a tensor of size 4 x 3 x 2
B.data =
(:,:,1) =
    0.8462    0.8381    0.8318
    0.5252    0.0196    0.5028
    0.2026    0.6813    0.7095
    0.6721    0.3795    0.4289
(:,:,2) =
    0.3046    0.3028    0.3784
    0.1897    0.5417    0.8600
    0.1934    0.1509    0.8537
    0.6822    0.6979    0.5936

ttt(A,B,[1 3],[2 3])

ans is a tensor of size 4 x 4
ans.data =
    1.7243    0.8943    1.0419    1.3295
    2.2316    1.1254    1.6976    1.7408
    1.5435    1.0523    1.2286    1.4355
    1.5717    0.9954    1.2404    1.7594
```

Figure 14. Computing the contracted product with “tensor times a tensor.”

```
A = tensor(rand(3,4,2))

A is a tensor of size 3 x 4 x 2
A.data =
(:,:,1) =
    0.4966    0.6449    0.3420    0.5341
    0.8998    0.8180    0.2897    0.7271
    0.8216    0.6602    0.3412    0.3093
(:,:,2) =
    0.8385    0.7027    0.6946    0.9568
    0.5681    0.5466    0.6213    0.5226
    0.3704    0.4449    0.7948    0.8801

B = tensor(rand(3,4,2))

B is a tensor of size 3 x 4 x 2
B.data =
(:,:,1) =
    0.1730    0.2523    0.1365    0.1991
    0.9797    0.8757    0.0118    0.2987
    0.2714    0.7373    0.8939    0.6614
(:,:,2) =
    0.2844    0.9883    0.5155    0.2259
    0.4692    0.5828    0.3340    0.5798
    0.0648    0.4235    0.4329    0.7604

ttt(A,B,[1:3])

ans is a tensor of order 0 (i.e., a scalar)
ans.data =
    7.2681
```

Figure 15. Computing the inner product with “tensor times a tensor.”

4 Matricize: Transforming a Tensor into a Matrix

It is often useful to rearrange the elements of a tensor so that they form a matrix. Although many names for this process exist, we call it "matricizing," following Kiers [9], because matricizing a tensor is analogous to vectorizing a matrix. De Lathauwer et al. [5] call this process "unfolding." It is sometimes also called "flattening" (see, e.g., [15]).

4.1 General Matricize

Let $\mathcal{A}$ be an $I_1 \times I_2 \times \cdots \times I_N$ tensor, and suppose we wish to rearrange it to be matrix of size $J_1 \times J_2$. Clearly, the number of entries in the matrix must be the same as the number of entries in the tensor; in other words, $\prod_{n=1}^{N} I_n = J_1 J_2$. Given J_1 and J_2 satisfying the above property, the mapping can be done any number of ways so long as we have a one-to-one mapping π such that

$$\pi : \{1, \ldots, I_1\} \times \{1, \ldots, I_2\} \times \cdots \times \{1, \cdots, I_N\} \to \{1, \ldots, J_1\} \times \{1, \ldots, J_2\}.$$

The `tensor_as_matrix` class supports the conversion of a tensor to a matrix as follows. Let the set of indices be partitioned into two disjoint subsets: $\{1, \ldots, N\} = \{r_1, \ldots, r_K\} \cup \{c_1, \ldots, c_L\}$. The set $\{r_1, \ldots, r_K\}$ defines those indices that will be mapped to the row indices of the resulting matrix and the set $\{c_1, \ldots, c_L\}$ defines those indices that will likewise be mapped to the column indices. In this case,

$$J_1 = \prod_{k=1}^{K} I_{r_k} \quad \text{and} \quad J_2 = \prod_{\ell=1}^{L} I_{c_\ell}.$$

Then we define $\pi(i_1, i_2, \ldots, i_N) = (j_1, j_2)$ where

$$j_1 = 1 + \sum_{k=1}^{K} \left[(i_{r_k} - 1) \prod_{\hat{k}=1}^{k-1} I_{r_{\hat{k}}} \right] \quad \text{and} \quad j_2 = 1 + \sum_{\ell=1}^{L} \left[(i_{r_\ell} - 1) \prod_{\hat{\ell}=1}^{\ell-1} I_{r_{\hat{\ell}}} \right].$$

Note that the sets $\{r_1, \ldots, r_K\}$ and $\{c_1, \ldots, c_L\}$ can be in any order and are not necessarily ascending. The following MATLAB commands convert a tensor to a matrix, and Figure 16 shows some examples.

A = `tensor_as_matrix(T,RDIMS)` matricizes $\mathcal{T}$ such that the dimensions (or modes) specified in `RDIMS` map to the rows of the matrix (in the order given), and the remaining dimensions (in ascending order) map to the columns.

A = `tensor_as_matrix(T,RDIMS,CDIMS)` matricizes $\mathcal{T}$ such that the dimensions specified in `RDIMS` map to the rows of the matrix, and the dimensions specified in `CDIMS` map to the columns, both in the order given.

A `tensor_as_matrix` object can be converted to a matrix as follows.

`B = double(A)` converts the `tensor_as_matrix` object to a matrix.

Also, the size of the corresponding tensor, the tensor indices corresponding to the matrix rows, and tensor indices corresponding to the matrix columns can be extracted as follows.

`sz = A.tsize` gives the size of the corresponding tensor.

`ridx = A.rindices` gives the indices that have been mapped to the rows, i.e., $\{r_1, \ldots, r_K\}$.

`cidx = A.cindices` gives the indices that have been mapped to the columns, i.e., $\{c_1, \ldots, c_L\}$.

With overloaded functions in MATLAB, the `tensor_as_matrix` class allows multiplication between tensors and/or matrices. More precisely, `mtimes(A,B)` is called for the syntax `A * B` when `A` or `B` is a `tensor_as_matrix` object. The result is another `tensor_as_matrix` object that can be converted back into a `tensor` object, as described below. The multiplication is analogous to the functionality provided by `ttt` for multiplying two `tensor` objects. Figure 17 shows an example of tensor-tensor multiplication using `tensor_as_matrix` objects.

Given a `tensor_as_matrix` object, we can automatically rearrange its entries back into a tensor by passing the `tensor_as_matrix` object into the constructor for the `tensor` class. The `tensor_as_matrix` class contains the mode of matricization and original tensor dimensions, making the conversion transparent to the user. The following MATLAB command can convert a matrix to a tensor, and Figure 18 shows such a conversion.

`tensor(A)` creates a tensor from `A`, which is a `tensor_as_matrix` object.

```
T = tensor(reshape(1:24,[4 3 2]))

T is a tensor of size 4 x 3 x 2
T.data =
(:,:,1) =
     1     5     9
     2     6    10
     3     7    11
     4     8    12
(:,:,2) =
    13    17    21
    14    18    22
    15    19    23
    16    20    24

% Matricizing the tensor
tensor_as_matrix(T,[1])

ans is a matrix corresponding to a tensor of size 4 x 3 x 2.
Row index spans tensor mode(s) [1].
Column index spans tensor mode(s) [2, 3].
ans.data =
     1     5     9    13    17    21
     2     6    10    14    18    22
     3     7    11    15    19    23
     4     8    12    16    20    24

% Another way of matricizing the tensor
tensor_as_matrix(T,[2])

ans is a matrix corresponding to a tensor of size 4 x 3 x 2.
Row index spans tensor mode(s) [2].
Column index spans tensor mode(s) [1, 3].
ans.data =
     1     2     3     4    13    14    15    16
     5     6     7     8    17    18    19    20
     9    10    11    12    21    22    23    24

% And yet another way of matricizing the tensor
tensor_as_matrix(T,[2],[3 1])

ans is a matrix corresponding to a tensor of size 4 x 3 x 2.
Row index spans tensor mode(s) [2].
Column index spans tensor mode(s) [3, 1].
ans.data =
     1    13     2    14     3    15     4    16
     5    17     6    18     7    19     8    20
     9    21    10    22    11    23    12    24
```

Figure 16. Matricizing a tensor using the `tensor_as_matrix` class.

```
% Create a random tensor and matricize it
T = rand(3,4,2);
A = tensor_as_matrix(T,2)

A is a matrix corresponding to a tensor of size 3 x 4 x 2.
Row index spans tensor mode(s) [2].
Column index spans tensor mode(s) [1, 3].
A.data =
    0.5390    0.1358    0.5949    0.1808    0.5269    0.9991
    0.5256    0.3757    0.4611    0.4731    0.5443    0.1961
    0.3962    0.0959    0.8326    0.8298    0.5672    0.1057
    0.7578    0.1490    0.1960    0.1779    0.6324    0.5835

% Compute the product of two matricized tensors
A' * A

ans is a matrix corresponding to a tensor of size 3 x 2 x 3 x 2.
Row index spans tensor mode(s) [1, 2].
Column index spans tensor mode(s) [3, 4].
ans.data =
    1.2980    0.4216    1.0414    0.8097    1.2740    1.1256
    0.4216    0.1910    0.3631    0.3084    0.4247    0.3064
    1.0414    0.3631    1.2982    1.0515    1.1606    0.8872
    0.8097    0.3084    1.0515    0.9767    0.9359    0.4649
    1.2740    0.4247    1.1606    0.9359    1.2955    1.0621
    1.1256    0.3064    0.8872    0.4649    1.0621    1.3883
```

Figure 17. Tensor-tensor multiplication using `tensor_as_matrix` objects.

```
% Convert the tensor_as_matrix object from the previous example
% into a tensor

T = tensor(A)
T is a tensor of size 3 x 4 x 2
T.data =
(:,:,1) =
    0.5390    0.5256    0.3962    0.7578
    0.1358    0.3757    0.0959    0.1490
    0.5949    0.4611    0.8326    0.1960
(:,:,2) =
    0.1808    0.4731    0.8298    0.1779
    0.5269    0.5443    0.5672    0.6324
    0.9991    0.1961    0.1057    0.5835
```

Figure 18. Constructing a `tensor` by reshaping a `tensor_as_matrix` object.

4.2 Mode-n Matricize

Typically, a tensor is matricized so that all of the fibers associated with a particular *single* dimension are aligned as the columns of the resulting matrix. In other words, we align the fibers of dimension n of a tensor $\mathcal{A}$ to be the columns of the matrix. This is a special case of the general matricize where only one dimension is mapped to the rows, so $K = 1$ and $\{r_1\} = \{n\}$. The resulting matrix is typically denoted by $\mathbf{A}_{(n)}$.

The columns can be ordered in any way. Two standard, but different, orderings are used by De Lathauwer et al. [5] and Kiers [9]. Both are cyclic, but the order is reversed. For De Lathauwer et al. [5], the ordering is given by $\{c_1, \ldots, c_L\} = \{n-1, n-2, \ldots, 1, N, N-1, \ldots, n+1\}$, and we refer to this ordering as *backward cyclic* or "bc" for short. For Kiers [9], the ordering is given by $\{c_1, \ldots, c_L\} = \{n+1, n+2, \ldots, N, 1, 2, \ldots, n-1\}$, and we refer to this ordering as *forward cyclic* or "fc" for short. Figure 19 shows the backward cyclic ordering and Figure 20 shows the forward cyclic ordering.

The following MATLAB commands can convert a tensor to a matrix according to the two definitions above, and Figure 21 shows two examples of matricizing a tensor.

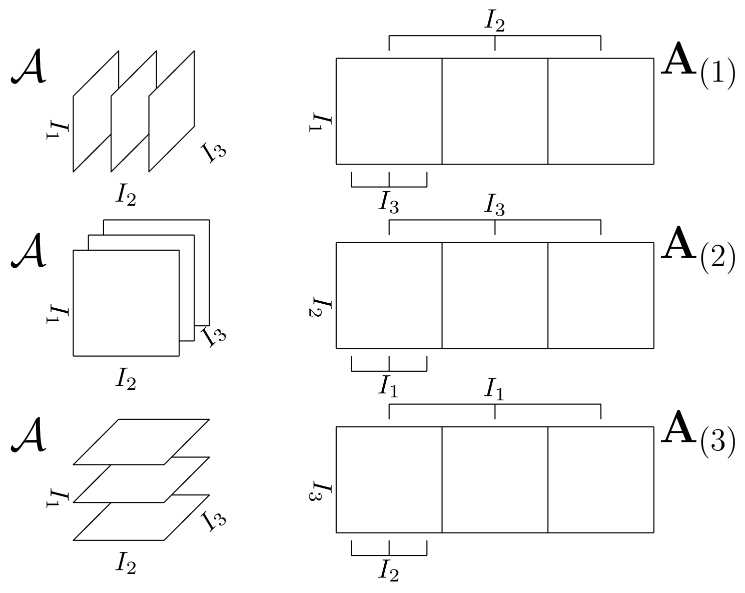

Figure 19. Backward cyclic matricizing a 3-way tensor.

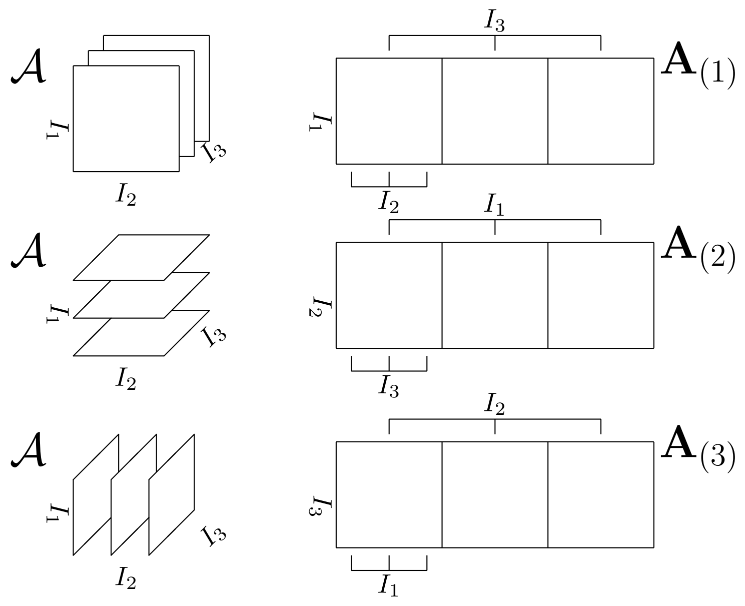

Figure 20. Forward cyclic matricizing a 3-way tensor.

`tensor_as_matrix(T,n,'bc')` computes $\mathbf{T}_{(n)}$, i.e., matricizing $\mathcal{T}$ using a backward cyclic ordering. The equivalent general command is `tensor_as_matrix(T,n, [n-1:-1:1 ndims(T):-1:n+1])`.

`tensor_as_matrix(T,n,'fc')` computes $\mathbf{T}_{(n)}$, i.e., matricizing $\mathcal{T}$ using a forward cyclic ordering. The equivalent general command is `tensor_as_matrix(T,n, [n+1:ndims(T) 1:n-1])`.

One benefit of matricizing is that tensors stored in matricized form may be manipulated as matrices, reducing n-mode multiplication, for example, to a matrix-matrix operation. If $\mathcal{B} = \mathcal{A} \times_n \mathbf{M}$, then

$$\mathbf{B}_{(n)} = \mathbf{M}\mathbf{A}_{(n)}.$$

Moreover, the series of n-mode products in (2), when written as a matrix formulation, can be expressed as a series of Kronecker products involving the $\mathbf{U}$ matrices. Consider the ordering of the tensor dimensions that map to the column space of the matrix (e.g., for forward cyclic ordering about mode-3 on a 4th-order tensor; then $\{r_1\} = \{3\}$ and $\{c_1, c_2, c_3\} = \{4, 1, 2\}$). The series of n-mode products in (2) is given by

$$\mathbf{B}_{(n)} = \mathbf{U}^{(n)}\mathbf{A}_{(n)} \left(\mathbf{U}^{(c_{n-1})} \otimes \mathbf{U}^{(c_{n-2})} \otimes \cdots \otimes \mathbf{U}^{(c_1)}\right)^T.$$

Figure 22 shows an example of computing the series of n-mode products using `tensor_as_matrix` and Kronecker products. The `tensor_as_matrix` object is converted into a standard MATLAB matrix for matrix-matrix multiplication, and the result must be converted back to a tensor with further matrix manipulations. Because this approach requires that the user code some lower level details, this example highlights the simplicity of the `tensor` class, which accomplishes the same computation in one function call to `ttm`.

5 Decomposed Tensors

As mentioned previously, we have also created two additional classes to support the representation of tensors in decomposed form, that is, as the sum of rank-1 tensors. A rank-1 tensor is a tensor that can be written as the outer product of vectors, i.e.,

$$\mathcal{A} = \lambda\, \mathbf{u}^{(1)} \circ \mathbf{u}^{(2)} \circ \cdots \circ \mathbf{u}^{(N)},$$

where λ is a scalar and each $\mathbf{u}^{(n)}$ is an I_n-vector, for $n = 1, \ldots, N$. The $\circ$ symbol denotes the outer product; so, in this case, the $(i_1, i_2, \ldots, i_N)$ entry of $\mathcal{A}$ is given by

$$\mathcal{A}(i_1, i_2, \ldots, i_N) = \lambda\, \mathbf{u}^{(1)}_{i_1}\, \mathbf{u}^{(2)}_{i_2} \cdots \mathbf{u}^{(N)}_{i_N},$$

where $\mathbf{u}_i$ denotes the ith entry of vector $\mathbf{u}$. We focus on two different tensor decompositions: CP and Tucker.

```
% Let T be a 3 x 4 x 2 tensor
T = tensor(rand(3,4,2))

T is a tensor of size 3 x 4 x 2
T.data =
(:,:,1) =
    0.5390    0.5256    0.3962    0.7578
    0.1358    0.3757    0.0959    0.1490
    0.5949    0.4611    0.8326    0.1960
(:,:,2) =
    0.1808    0.4731    0.8298    0.1779
    0.5269    0.5443    0.5672    0.6324
    0.9991    0.1961    0.1057    0.5835

% Backward cyclic
A1 = tensor_as_matrix(T,2,'bc')

A1 is a matrix corresponding to a tensor of size 3 x 4 x 2.
Row index spans tensor mode(s) [2].
Column index spans tensor mode(s) [1, 3].
A1.data =
    0.5390    0.1358    0.5949    0.1808    0.5269    0.9991
    0.5256    0.3757    0.4611    0.4731    0.5443    0.1961
    0.3962    0.0959    0.8326    0.8298    0.5672    0.1057
    0.7578    0.1490    0.1960    0.1779    0.6324    0.5835

% Forward cyclic
A2 = tensor_as_matrix(T,2,'fc')

A2 is a matrix corresponding to a tensor of size 3 x 4 x 2.
Row index spans tensor mode(s) [2].
Column index spans tensor mode(s) [3, 1].
A2.data =
    0.5390    0.1808    0.1358    0.5269    0.5949    0.9991
    0.5256    0.4731    0.3757    0.5443    0.4611    0.1961
    0.3962    0.8298    0.0959    0.5672    0.8326    0.1057
    0.7578    0.1779    0.1490    0.6324    0.1960    0.5835
```

Figure 21. Two choices for converting a tensor to a matrix.

```
% Compute a random T and a sequence of matrices
sizeT = [5,4,3,2];
T = tensor(rand(sizeT));
for n = 1:ndims(T)
   U{n} = rand(size(T,n),size(T,n));
end

% Use a tensor command to compute the
% tensor times the sequence of matrices
A = ttm(T,U);

% Manipulate the tensor instead in matrix form
% to compute the same result.
rdim = 2;
M = tensor_as_matrix(T, rdim, 'fc');
cdims = M.cindices;
B = U{rdim} * double(M);
B = B * kron(kron(U{cdims(3)},U{cdims(2)}),U{cdims(1)})';

% Reshape result to match A
B = reshape(B, [sizeT(rdim) sizeT(cdims)] );
[sdims sindx] = sort([rdim cdims]);
B = permute(B,sindx);

% Compare the results
val = norm(A - tensor(B)) / norm(A)

val =
   1.5292e-16
```

Figure 22. Example of computing the series of n-mode products using `tensor_as_matrix`.

5.1 CP tensors

Recall that "CP" is shorthand for CANDECOMP [2] and PARAFAC [7], which are identical decompositions that were developed independently for different applications. The CP decomposition is a weighted sum of rank-1 tensors, given by

$$\mathcal{A} = \sum_{k=1}^{K} \lambda_k \, \mathbf{U}^{(1)}_{:k} \circ \mathbf{U}^{(2)}_{:k} \circ \cdots \circ \mathbf{U}^{(N)}_{:k}. \tag{4}$$

Here λ is a vector of size K and each $\mathbf{U}^{(n)}$ is a matrix of size $I_n \times K$, for $n = 1, \ldots, N$. Recall that the notation $\mathbf{U}^{(n)}_{:k}$ denotes the kth column of the matrix $\mathbf{U}^{(n)}$.

The following MATLAB command creates a CP tensor.

`T = cp_tensor(lambda,U)` creates a `cp_tensor` object. Here `lambda` is a K-vector and `U` is a cell array whose nth entry is the matrix $U^{(n)}$ with K columns.

A CP tensor can be converted to a dense tensor as follows; see Figure 23 for an example.

`B = full(A)` converts a `cp_tensor` object to a `tensor` object.

Addition and subtraction of CP tensors is handled in a special manner. The λ's and $\mathbf{U}^{(n)}$'s are concatenated. To add or subtract two CP tensors (of the same order and size), use the `+` and `-` signs. An example is shown in Figure 24.

`A + B` computes the sum of two CP tensors.

`A - B` computes the difference of two CP tensors.

To determine the value of K for a CP tensor, execute the following MATLAB command.

`r = length(T.lambda)` returns the "rank" of the tensor `T`.

```
A = cp_tensor(5, [2 3 4]', [1 2]', [5 4 3]')

A is a CP tensor of size 3 x 2 x 3
A.lambda =
     5
A.U{1} =
     2
     3
     4
A.U{2} =
     1
     2
A.U{3} =
     5
     4
     3

% The size of A
size(A)

ans =
     3     2     3

% The ``rank'' of A
length(A.lambda)

ans =
     1

% Convert to a (dense) tensor
B = full(A)

B is a tensor of size 3 x 2 x 3
B.data =
(:,:,1) =
    50   100
    75   150
   100   200
(:,:,2) =
    40    80
    60   120
    80   160
(:,:,3) =
    30    60
    45    90
    60   120
```

Figure 23. Creating a CP tensor.

```
A = cp_tensor(5, [2 3 4]', [1 2]', [5 4 3]');
B = A + A

B is a CP tensor of size 3 x 2 x 3
B.lambda =
     5
     5
B.U{1} =
     2     2
     3     3
     4     4
B.U{2} =
     1     1
     2     2
B.U{3} =
     5     5
     4     4
     3     3

C = full(B)

C is a tensor of size 3 x 2 x 3
C.data =
(:,:,1) =
   100   200
   150   300
   200   400
(:,:,2) =
    80   160
   120   240
   160   320
(:,:,3) =
    60   120
    90   180
   120   240
```

Figure 24. Adding two CP tensors.

5.2 Tucker tensors

The Tucker decomposition [14], also called a Rank-$(K_1, K_2, \ldots, K_N)$ decomposition [6], is another way of summing decomposed tensors and is given by

$$\mathcal{A} = \sum_{k_1=1}^{K_1} \sum_{k_2=1}^{K_2} \cdots \sum_{k_N=1}^{K_N} \lambda(k_1, k_2, \ldots, k_N)\, \mathbf{U}^{(1)}_{:k_1} \circ \mathbf{U}^{(2)}_{:k_2} \circ \cdots \circ \mathbf{U}^{(N)}_{:k_N}. \tag{5}$$

Here λ is itself a tensor of size $K_1 \times K_2 \times \cdots \times K_N$, and each $\mathbf{U}^{(n)}$ is a matrix of size $I_n \times K_n$, for $n = 1, \ldots, N$. As before, the notation $\mathbf{U}^{(n)}_{:k}$ denotes the kth column of the matrix $\mathbf{U}^{(n)}$. The tensor λ is often called the "core array" or "core tensor."

A Tucker tensor can be created in MATLAB as follows; Figure 25 shows an example.

`T = tucker_tensor(lambda,U)` where `lambda` is a $K_1 \times K_2 \times \cdots \times K_N$ tensor and `U` is a cell array whose nth entry is a matrix with K_n columns.

A Tucker tensor can be converted to a dense tensor as follows.

`B = full(A)` converts a `tucker_tensor` object to a `tensor` object.

5.3 Relationship between CP and Tucker tensors

Mathematically, a CP decomposition is a special case of a Tucker decomposition where $K = K_1 = K_2 = \cdots = K_N$ and $\lambda(k_1, k_2, \ldots, k_N)$ is zero unless $k_1 = k_2 = \cdots = k_N$ (i.e., only the diagonal entries of the tensor λ are non-zero). On the other hand, it is possible to express a Tucker decomposition as a CP decomposition where $K = \prod_{n=1}^{N} K_n$.

6 Examples

We demonstrate the use of the `tensor`, `cp_tensor`, and `tucker_tensor` classes for algorithm development by implementing the higher-order generalizations of the power method and orthogonal iteration presented by De Lathauwer et al. [6].

```
lambda = tensor(rand(4,3,1),[4 3 1]);
for n = 1 : 3
    U{n} = rand(5,size(lambda,n));
end
A = tucker_tensor(lambda,U)

A is a Tucker tensor of size 5 x 5 x 5
A.lambda =
Tensor of size 4 x 3 x 1
data =
    0.8939    0.2844    0.5828
    0.1991    0.4692    0.4235
    0.2987    0.0648    0.5155
    0.6614    0.9883    0.3340
A.U{1} =
    0.4329    0.6405    0.4611    0.0503
    0.2259    0.2091    0.5678    0.4154
    0.5798    0.3798    0.7942    0.3050
    0.7604    0.7833    0.0592    0.8744
    0.5298    0.6808    0.6029    0.0150
A.U{2} =
    0.7680    0.4983    0.7266
    0.9708    0.2140    0.4120
    0.9901    0.6435    0.7446
    0.7889    0.3200    0.2679
    0.4387    0.9601    0.4399
A.U{3} =
    0.9334
    0.6833
    0.2126
    0.8392
    0.6288

% The size of A
size(A)
ans =
     5     5     5

% The "rank" of A
size(A.lambda)
ans =
     4     3     1
```

Figure 25. Creating a Tucker tensor.

The first example is the higher-order power method, Algorithm 3.2 of De Lathauwer et al. [6], which is a multilinear generalization of the best rank-1 approximation problem for matrices. The best rank-1 approximation problem is that, given a tensor $\mathcal{A}$, we want to find a $\mathcal{B}$ of the form

$$B = \lambda\, \mathbf{u}^{(1)} \circ \mathbf{u}^{(2)} \circ \cdots \circ \mathbf{u}^{(N)},$$

such that $\|\, \mathcal{A} - \mathcal{B} \,\|$ is as small as possible. The higher-order power method computes a $\mathcal{B}$ that approximately solves this problem. Essentially, this method works as follows. It fixes all $\mathbf{u}$-vectors except $\mathbf{u}^{(1)}$ and then solves for the optimal $\mathbf{u}^{(1)}$, likewise for $\mathbf{u}^{(2)}$, $\mathbf{u}^{(3)}$, and so on, cycling through the indices until the specified number of iterations is exhausted. In Figure 26, we show the algorithm using our new notation, Figure 27 shows the MATLAB code that implements the algorithm, and Figure 28 shows sample output.

The second example is the higher-order orthogonal iteration, which is the multilinear generalization of the best rank-R approximation problem for matrices. Algorithm 4.2 in [6] is the higher-order orthogonal iteration and finds the best rank-$(R_1, R_2, \ldots, R_N)$ approximation of a higher-order tensor. We have reproduced this algorithm in Figure 29 using our new notation. Our corresponding MATLAB implementation is listed in Figure 30, and Figures 31–33 show the computation of different rank-(R_1, R_2, R_3) approximations to the same random tensor from Figure 28.

7 Conclusions

We have described four new MATLAB classes for manipulating dense and factored tensors. These classes extend MATLAB's built-in capabilities for multidimensional arrays in order to facilitate rapid algorithm development when dealing with tensor datatypes.

The `tensor` class simplifies the algorithmic details for implementing numerical methods for higher-order tensors by hiding the underlying matrix operations. It was previously the case that users had to know how to appropriately reshape the tensor into a matrix, execute the desired operation using matrix commands, and then appropriately reshape the result into a tensor. This can be nonintuitive and cumbersome, and we believe using the `tensor` class will be much simpler.

The `tensor_as_matrix` class offers a way to convert a higher-order tensor into a matrix. Many existing algorithms in the literature that deal with tensors rely on matrix-matrix operations. The `tensor_as_matrix` functionality offers a means to implement these algorithms more easily, without the difficulty of reshaping and permuting tensor objects to the desired shape.

The `tucker_tensor` and `cp_tensor` classes give users an easy way to store and

```
HIGHER-ORDER POWER METHOD
In: $\mathcal{A}$ of size $I_1 \times I_2 \times \cdots \times I_N$.
Out: $\mathcal{B}$ of size $I_1 \times I_2 \times \cdots \times I_N$, an estimate of the best rank-1
approximation of $\mathcal{A}$.

  1. Compute initial values: Let $\mathbf{u}_0^{(n)}$ be the dominant left singular
     vector of $\mathbf{A}_{(n)}$ for $n = 2, \ldots, N$.
  2. For $k = 0, 1, 2, \ldots$ (until converged), do:
        For $n = 1, \ldots, N$, do:
            $\tilde{\mathbf{u}}_{k+1}^{(n)} = \mathcal{A} \,\bar{\times}_{-n}\, \{\mathbf{u}_k\}$.
            $\lambda_{k+1}^{(n)} = \left\| \tilde{\mathbf{u}}_{k+1}^{(n)} \right\|$
            $\mathbf{u}_{k+1}^{(n)} = \tilde{\mathbf{u}}_{k+1}^{(n)} / \lambda_{k+1}^{(n)}$
  3. Let $\lambda = \lambda_K$ and $\{\mathbf{u}\} = \{\mathbf{u}_K\}$, where $K$ is the index of the final
     result of step 2.
  4. Set $\mathcal{B} = \lambda \, \mathbf{u}^{(1)} \circ \mathbf{u}^{(2)} \circ \cdots \circ \mathbf{u}^{(n)}$.
```

Figure 26. Higher-order power method algorithm of De Lathauwer, De Moor, and Vandewalle using the proposed notation. In this illustration, subscripts denote iteration number.

```
function B = hopm(A,kmax)

A = tensor(A);
N = ndims(A);

% Default value
if ~exist('kmax','var')
    kmax = 5;
end

% Compute the dominant left singluar vectors
% of A_(n) (2 <= n <= N)
for n = 2:N
  [u{n}, lambda(n), V] = ...
     svds(double(tensor_as_matrix(A,n)), 1);
end

% Iterate until convergence
for k = 1:kmax
  for n = 1:N
    u{n} = ttv(A, u, -n);
    lambda(n) = norm(U{n});
    u{n} = double(u{n}./lambda(n));
  end
end

% Assemble the resulting tensor
B = cp_tensor(lambda(N), u);
```

Figure 27. MATLAB code for our implementation of the higher-order power method.

```
T = tensor(rand(3,4,2))

T is a tensor of size 3 x 4 x 2
T.data =
(:,:,1) =
    0.8030    0.9159    0.8735    0.4222
    0.0839    0.6020    0.5134    0.9614
    0.9455    0.2536    0.7327    0.0721
(:,:,2) =
    0.5534    0.3358    0.3567    0.5625
    0.2920    0.6802    0.4983    0.6166
    0.8580    0.0534    0.4344    0.1133

T1 = hopm(T)

T1 is a CP tensor of size 3 x 4 x 2
T1.lambda =
    2.6206
T1.U{1} =
   -0.6717
   -0.5446
   -0.5023
T1.U{2} =
    0.5431
    0.4700
    0.5444
    0.4333
T1.U{3} =
   -0.8075
   -0.5899

T1f = full(T1)

T1f is a tensor of size 3 x 4 x 2
T1f.data =
(:,:,1) =
    0.7719    0.6680    0.7738    0.6159
    0.6258    0.5416    0.6274    0.4993
    0.5772    0.4996    0.5787    0.4606
(:,:,2) =
    0.5639    0.4880    0.5653    0.4499
    0.4572    0.3956    0.4583    0.3647
    0.4217    0.3649    0.4227    0.3364

(norm(T) - norm(T1f)) / norm(T)

ans =
    9.9110e-2
```

Figure 28. Example of the higher-order power method.

```
HIGHER-ORDER ORTHOGONAL ITERATION
In: $\mathcal{A}$ of size $I_1 \times I_2 \times \cdots \times I_N$ and desired rank of output.
Out: $\mathcal{B}$ of size $I_1 \times I_2 \times \cdots \times I_N$, an estimate of the best
rank-$(R_1, R_2, \ldots, R_N)$ approximation of $\mathcal{A}$.

  1. Compute initial values: Let $\mathbf{U}_0^{(n)} \in \mathbb{R}^{I_n \times R_n}$ be an orthonormal
     basis for the dominant $R_n$-dimensional left singular subspace of
     $\mathbf{A}_{(n)}$ for $n = 2, \ldots, n$.
  2. For $k = 0, 1, 2, \ldots$ (until converged), do:
       For $n = 1, \ldots, N$, do:
           $\tilde{\mathcal{U}} = \mathcal{A} \times_{-n} \{\mathbf{U}_k^T\}$
           Let $\mathbf{W}$ of size $I_n \times R_n$ solve:
           $\max \left\| \tilde{\mathcal{U}} \times_n \mathbf{W}^T \right\|$ subject to $\mathbf{W}^T\mathbf{W} = \mathbf{I}$.
           $\mathbf{U}_{k+1}^{(n)} = \mathbf{W}$.
  3. Let $\{\mathbf{U}\} = \{\mathbf{U}_K\}$, where $K$ is the index of the final result of step 2.
  4. Set $\lambda = \mathcal{A} \times \{\mathbf{U}^T\}$.
  5. Set $\mathcal{B} = \lambda \times \{\mathbf{U}\}$.
```

Figure 29. Higher-order orthogonal iteration algorithm of De Lathauwer, De Moor, and Vandewalle using the proposed notation. In this illustration, subscripts denote iteration number.

```
function B = hooi(A,R,kmax)

A = tensor(A);
N = ndims(A);

% Default value
if ~exist('kmax','var')
    kmax = 5;
end

% Compute an orthonormal basis for the dominant
% Rn-dimensional left singular subspace of
% A_(n) (1 <= n <= N).  We store its transpose.
for n = 1:N
  [U, S, V] = ...
    svds(double(tensor_as_matrix(A,n)), R(n));
  Ut{n} = U';
end

% Iterate until convergence
for k = 1:kmax
  for n = 1:N
    Utilde = ttm(A, Ut, -n);

    % Maximize norm(Utilde x_n W') wrt W and
    % keeping orthonormality of W
    [W,S,V] = ...
      svds(double(tensor_as_matrix(Utilde, n)), R(n));
    Ut{n} = W';
  end
end

% Create the core array
lambda = ttm(A, Ut);

% Create cell array containing U from the cell
% array containing its transpose
for n = 1:N
  U{n} = Ut{n}';
end

% Assemble the resulting tensor
B = tucker_tensor(lambda, U);
```

Figure 30. MATLAB code for our implementation of the higher-order orthogonal iteration method.

```
T2 = hooi(T,[1 1 1])

T2 is a Tucker tensor of size 3 x 4 x 2
T2.lambda =
Tensor of size 1 x 1 x 1
data =
    2.6206
T2.U{1} =
    0.6717
    0.5446
    0.5023
T2.U{2} =
   -0.5431
   -0.4700
   -0.5444
   -0.4333
T2.U{3} =
   -0.8075
   -0.5899

T2f = full(T2)

T2f is a tensor of size 3 x 4 x 2
T2f.data =
(:,:,1) =
    0.7719    0.6680    0.7738    0.6159
    0.6258    0.5416    0.6274    0.4993
    0.5772    0.4996    0.5787    0.4606
(:,:,2) =
    0.5639    0.4880    0.5653    0.4499
    0.4572    0.3956    0.4583    0.3647
    0.4217    0.3649    0.4227    0.3364

norm(T2f)

ans =
    2.6206
```

Figure 31. Example of the higher-order orthogonal iteration for computing the best rank-(1,1,1) tensor.

```
T3 = hooi(T,[2 2 1])

T3 is a Tucker tensor of size 3 x 4 x 2
T3.lambda =
Tensor of size 2 x 2 x 1
data =
   -2.6206    0.0000
   -0.0000   -1.1233
T3.U{1} =
    0.6718   -0.0186
    0.5445    0.6903
    0.5023   -0.7233
T3.U{2} =
    0.5430   -0.6862
    0.4701    0.3773
    0.5445   -0.1259
    0.4332    0.6090
T3.U{3} =
   -0.8083
   -0.5888

T3f = full(T3)

T3f is a tensor of size 3 x 4 x 2
T3f.data =
(:,:,1) =
    0.7843    0.6625    0.7769    0.6061
    0.1962    0.7786    0.5491    0.8813
    1.0284    0.2523    0.6620    0.0610
(:,:,2) =
    0.5713    0.4826    0.5659    0.4415
    0.1429    0.5671    0.3999    0.6419
    0.7491    0.1838    0.4822    0.0444

norm(T3f)

ans =
    2.8512
```

Figure 32. Example of the higher-order orthogonal iteration for computing the best rank-(2,2,1) tensor.

```
T4 = hooi(T,[3 4 2])

T4 is a Tucker tensor of size 3 x 4 x 2
T4.lambda =
Tensor of size 3 x 4 x 2
data =
(:,:,1) =
    2.6201   -0.0075   -0.0338   -0.0029
   -0.0176    1.1198   -0.0118   -0.0006
    0.0142    0.0854   -0.1634   -0.1214
(:,:,2) =
   -0.0126   -0.0529    0.2446   -0.1165
    0.1747    0.0187    0.2369    0.0055
   -0.2249   -0.1523   -0.2324   -0.0311
T4.U{1} =
    0.6774    0.0720   -0.7321
    0.5344   -0.7321    0.4224
    0.5055    0.6774    0.5343
T4.U{2} =
    0.5442    0.6798   -0.2895    0.3974
    0.4774   -0.3606    0.6648    0.4474
    0.5474    0.1200    0.2110   -0.8009
    0.4200   -0.6272   -0.6556    0.0203
T4.U{3} =
    0.8117    0.5841
    0.5841   -0.8117

T4f = full(T4)

T4f is a tensor of size 3 x 4 x 2
T4f.data =
(:,:,1) =
    0.8030    0.9159    0.8735    0.4222
    0.0839    0.6020    0.5134    0.9614
    0.9455    0.2536    0.7327    0.0721
(:,:,2) =
    0.5534    0.3358    0.3567    0.5625
    0.2920    0.6802    0.4983    0.6166
    0.8580    0.0534    0.4344    0.1133

norm(T4f)

ans =
    2.9089
```

Figure 33. Example of the higher-order orthogonal iteration for computing the best rank-(3,4,2) tensor.

manipulate factored tensors, as well as the ability to convert such tensors into non-factored (or dense) format.

At this stage, our MATLAB implementations are not optimized for performance or memory usage; however, we have striven for consistency and ease-of-use. In the future, we plan to further enhance these classes and add additional functionality.

Over the course of this code development effort, we have relied on published notation, especially from Kiers [9] and De Lathauwer et al. [6]. To address ambiguities that we discovered in the class development process, we have proposed extensions to the existing mathematical notation, particularly in the area of tensor multiplication, that we believe more clearly denote mathematical concepts that were difficult to write succinctly with the existing notation.

We have demonstrated our new notation and MATLAB classes by revisiting the higher-order power method and the higher-order orthogonal iteration method from [6]. In our opinion, the resulting algorithm and code is more easily understood using our consolidated notation and MATLAB classes.

Acknowledgments

For their many helpful suggestions, we gratefully acknowledge the participants at the Tensor Decompositions Workshop held at the American Institute of Mathematics (AIM) in Palo Alto, CA on July 19–23, 2004. We also thank AIM and the National Science Foundation for sponsoring the workshop.

References

[1] C. A. Andersson and R. Bro. The N-way toolbox for MATLAB. *Chemometrics & Intelligent Laboratory Systems*, 52(1):1–4, 2000. `http://www.models.kvl.dk/source/nwaytoolbox/`.

[2] J. D. Carroll and J. J. Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of 'Eckart-Young' decomposition. *Psychometrika*, 35:283–319, 1970.

[3] B. Chen, A. Petropolu, and L. De Lathauwer. Blind identification of convolutive mim systems with 3 sources and 2 sensors. *Applied Signal Processing (Special Issue on Space-Time Coding and Its Applications, Part II)*, (5):487–496, 2002.

[4] Pierre Comon. Tensor decompositions. In J. G. McWhirter and I. K. Proudler, editors, *Mathematics in Signal Processing V*, pages 1–24, Oxford, UK, 2001. Oxford University Press.

[5] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl.*, 21(4):1253–1278, 2000.

[6] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. On the best rank-1 and rank-$(R_1, R_2, \ldots, R_N)$ approximation of higher-order tensors. *SIAM J. Matrix Anal. Appl.*, 21(4):1324–1342, 2000.

[7] Richard A. Harshman. Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multi-modal factor analysis. *UCLA working papers in phonetics*, 16:1–84, 1970.

[8] Richard A. Harshman. An index formulism that generalizes the capabilities of matrix notation and algebra to n-way arrays. *J. Chemometrics*, 15:689–714, 2001.

[9] Henk A. L. Kiers. Towards a standardized notation and terminology in multiway analysis. *J. Chemometrics*, 14:105–122, 2000.

[10] Pieter Kroonenberg. Applications of three-mode techniques: Overview, problems, and prospects. Presentation at the 2004 AIM Tensor Decompositions Workshop, Palo Alto, California, July 19-23, 2004, 2004. `http://csmr.ca.sandia.gov/~tgkolda/tdw2004/Kroonenberg%20-%20Talk.pdf`.

[11] Age Smilde, Rasmus Bro, and Paul Geladi. *Multi-way Analysis: Applications in the Chemical Sciences*. Wiley, 2004.

[12] The MathWorks, Inc. Documentation: MATLAB: Programming: Classes and objects. `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/ch11_mat.html`, 2004.

[13] The MathWorks, Inc. Documentation: MATLAB: Programming: Multidimensional arrays. `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/ch_dat32.html#39663`, 2004.

[14] Ledyard R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31:279–311, 1966.

[15] M. A. O. Vasilescu and D. Terzopoulos. Multilinear analysis of image ensembles: Tensorfaces. In *Proc. 7th European Conference on Computer Vision (ECCV'02), Copenhagen, Denmark, May 2002*, volume 2350 of *Lecture Notes in Computer Science*, pages 447–460. Springer-Verlag, 2002.