

The **TenTS** gretl package

Giovanni Ferretti
Università Politecnica delle Marche
`g.ferretti@pm.univpm.it`

June 23, 2026

Abstract

The gretl package **TenTS** provides an easy way to handle tensor-valued time series (hence the name). Since the scripting language of gretl lacks the ability to define multi-dimensional arrays, a few utility functions are devoted to the creation and manipulation of N -index tensors.

The package, however, also contains functions for handling tensors with a special structure, in which one of the dimensions can be interpreted as time. A few applications are illustrated.

1 Introduction

The increasing availability of high-dimensional datasets has spurred considerable interest in the modelling of time series of objects whose internal structure is more complex than scalars or vectors (as in ARIMA/VARMA modelling and similar).

One such notable extension is the analysis of tensor-valued time series. A *tensor* of order N is a multi-dimensional array with N dimensions, or indices. That is,

$$\mathcal{X}_{i_1, i_2, \dots, i_N} = a$$

where a is a scalar; therefore, vectors and matrices can then be seen as order-1 and order-2 tensors, respectively.

For example, one may define a sequence of matrices through time as the object $\mathcal{X}_{i,j,t}$, where you have rows, columns and time period, but of course generalisation is obvious. Due to their flexibility, tensors are widely adopted in numerous fields, including econometrics, data analysis, machine learning, signal processing, and scientific computing. In econometrics, tensors have been used in several many contexts, such as Tensor Factor Models (TFM) [Chen et al., 2022b], [Han et al., 2024] and Tensor Autoregressive Models (TenAR) [Li and Xiao, 2021].

Since tensors are not natively supported as a **gretl** data type, this package provides a general framework for working with such multi-dimensional objects. In section 2 we outline the main ideas underlying tensor representation and manipulation in **TenTS**. In section 3 we describe three relevant mathematical operations involving tensors. In section 4 we describe some utilities for tensor valued time series that are supported by this package. We also describe TenAR models, as proposed in Li and Xiao [2021], and how they are implemented in

the **TenTS** package. A complete description of the functions provided by the package is given in the Section 5.

2 Representation of a tensor

The dimensions of a tensor can be thought of in a hierarchical way. For instance, consider the following order-3 tensor $\mathcal{X} \in \mathbb{R}^{J \times C \times K}$:

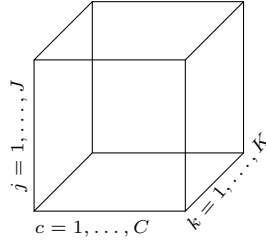


Figure 1: A third-order tensor: $\mathcal{X} \in \mathbb{R}^{J \times C \times K}$.

Loosely speaking, we can think of $\mathcal{X}$ as a container for a number of matrices equal to K , each of which has J rows and C columns, but of course the order of the dimensions can be exchanged without loss of generality, in what would be a straightforward generalisation of matrix transposition.

In line with this reasoning, in **gretl** a 3-order tensor $\mathcal{X}$ could be represented as follows:

```

scalar J = 4
scalar C = 3
scalar K = 10
matrices X = array(K)

loop k = 1 .. K
    X[k] = mnormal(J,C)
endloop
```

However, this would be difficult to generalise to tensors of order 4 or more.

Therefore, this package represents tensors in an abstract way that handles an arbitrary number of dimensions, via a bundle that contains an element called **tensor**; this will be a matrix for orders 1 and 2, a matrix array for order 3 (that is, an object of type **matrices**) and an array of arrays for higher orders.

The main idea is that the user doesn't have to worry about these differences, since the package functions handle all cases in the same way, thus providing a convenient level of abstraction. Depending on the tensor order, an element of the appropriate data type can be directly accessed via the **tensor** bundle key.

3 Three important Tensor Operations

3.1 Tensorizing a matrix

The easiest way to create a tensor is to define an appropriately-sized matrix and then fill the tensor bundle with the appropriate elements. This function is

similar in spirit to the native function `msplitby`, which creates a matrix array from a matrix.

Suppose you want to create a tensor of order 4 filled with uniform pseudo-random variates, that is

$$\mathcal{X}_{i,j,k,l} \sim U(0,1);$$

suppose also that the first index i goes from 1 to 4, j and k go from 1 to 2, and l goes from 1 to 3. Therefore, the total number of elements contained in $\mathcal{X}$ is 48. To build such a tensor, the `tensorize` function can be called as

```
bundle tX = tensorize(muniform(48,1), {4,2,2,3})
```

Printing `tX` would yield

```
bundle tX
  dims (matrix: 1 x 4)
  tensor = array of arrays, length 3
```

Since 4 dimensions were specified, the bundle `tX` will contain an element of type array of arrays under the key `tensor`. Therefore, in this case a single matrix can be accessed by fixing specific values for the fourth and third dimensions. For example, `tX.tensor[2][1]` selects the value 2 for the fourth dimension, and 1 for the third one, yielding something like:

```
tX.tensor[2][1]
```

```
0.31594    0.15505
0.16293    0.98205
0.24350    0.20373
0.028786   0.043227
```

To print out the entire tensor, the package provides the `tensor.print` function. More examples will be given in section 5.1.1. Under the bundle key `dims`, a row vector with a number of columns equal to the order of the tensor can be retrieved. Each element of this vector contains the total size of the respective dimension of the tensor, hence:

```
tX.dims
```

```
4    2    2    3
```

3.2 Matricizing a tensor

Matricization, also known as unfolding or flattening, is the process of reordering an order N tensor into a matrix [Kolda and Bader, 2009]. An order N tensor can be unfolded along any of its n -th dimensions $n = 1, \dots, N$. The resulting matrix is known as the n -th mode of the tensor, and has a number of rows equal to the size of its n -th dimension, and a number of columns equal to the product of the sizes of the other dimensions.¹

For instance, consider a tensor of order 3, $\mathcal{X} \in \mathbb{R}^{J \times C \times K}$. In this case we have three possible matricizations, depending on the selected dimension to unfold. These are: $\mathbf{X}_1 \in \mathbb{R}^{J \times (CK)}$, $\mathbf{X}_2 \in \mathbb{R}^{C \times (JK)}$, $\mathbf{X}_3 \in \mathbb{R}^{K \times (JC)}$. The `TenTS` package function which performs this operation is `unfold`. Here is an example:

¹For a complete description of the unfolding process, see Kolda and Bader [2009], pag. 459.

```

scalar J = 4
scalar C = 3
scalar K = 2
bundle X = tensorize(seq(1,J*C*K),{J,C,K})
tensor_print(X)

```

which gives

X[1]

1	5	9
2	6	10
3	7	11
4	8	12

X[2]

13	17	21
14	18	22
15	19	23
16	20	24

And here are its three possible matricizations:

```
eval unfold(X,1)
```

1	5	9	13	17	21
2	6	10	14	18	22
3	7	11	15	19	23
4	8	12	16	20	24

```
eval unfold(X,2)
```

1	2	3	4	13	14	15	16
5	6	7	8	17	18	19	20
9	10	11	12	21	22	23	24

```
eval unfold(X,3)
```

1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24

3.3 Tensor-to-Matrix Products

The Tensor to matrix operation is intimately linked to the unfolding process, and is also known as tensor n -mode product; see Kolda and Bader [2009], pag. 460 for a technical review of the properties of this operation.

In general, if an order- N tensor is multiplied by a matrix in the n -th mode, the resulting tensor still has N dimensions, but the size of the n -th one is modified. For instance, given the tensor $\mathcal{X} \in \mathbb{R}^{k_1, k_2, \dots, k_n, \dots, k_N}$, n -th mode product of $\mathcal{X}$ for a matrix $\mathbf{A} \in \mathbb{R}^{j \times k_n}$, is the following:

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

$$\mathcal{Y} \in \mathbb{R}^{k_1, k_2, \dots, k_{n-1}, j, k_{n+1}, \dots, k_N}$$

The connection with the unfolding operation is in the following property:

$$\mathbf{Y}_n = \mathbf{A} \mathbf{X}_n$$

meaning that the matricization of $\mathcal{Y}$ along the n -th dimension is equal to the product between the matrix $\mathbf{A}$ and the tensor $\mathcal{X}$ unfolded along the n -th dimension.

The multilinear product is a generalization of the n -mode product, where the tensor $\mathcal{X}$ is multiplied by a set of N matrices, one per each dimension of the tensor. The result tensor still has N dimensions, but all sizes may be modified. For example:

$$\mathcal{Y} = \mathcal{X} \times_1 \mathbf{A}_1 \times_2 \mathbf{A}_2 \times_3 \dots \times_N \mathbf{A}_N$$

Again, with the unfolding operator the multilinear product can be expressed as:

$$\mathbf{Y}_1 = \mathbf{A}_1 \mathbf{X}_1 (\mathbf{A}_N \otimes \mathbf{A}_{N-1} \otimes \dots \otimes \mathbf{A}_2)' \quad (1)$$

where $\otimes$ is the Kronecker product.

In **TenTS** both the n -mode product and the multilinear product are performed through the function `tensor_times_matrices`, which implements equation (1). In the former case matrices $\mathbf{A}_j, j \neq n$ are simply set as identity matrices of dimensions $k_j \times k_j$. Examples for both cases are given under the guide for `tensor_times_matrices`.

4 Tensor-valued time series

4.1 Basic Time-series Operations

A time series of N -order tensors can be represented in many ways: a particularly convenient one is by defining a $(N + 1)$ -order tensor in which one of the dimensions is time. **TenTS** contains several functions to handle this special type of tensors, so that common time-series operations such as computing lags and differences or producing plots can be readily performed via appropriate functions. Functions for tensor-valued time series, such as `tensor_lag` and `tensor_diff`, require users to specify which tensor dimension corresponds to time. This information is provided through the `time_id` bundle key.

As an example, consider a matrix-valued time series represented as a third-order tensor $\mathcal{Z} \in \mathbb{R}^{2 \times 2 \times T}$, where the third dimension corresponds to time. Assume that the elements of the series satisfy

$$z_{i,j,t} \sim \mathcal{N}(0, 1), \quad i, j = 1, 2, \quad t = 1, \dots, T.$$

That is, each mode-3 fiber Kolda and Bader [2009] of $\mathcal{Z}$ is a white-noise process. Now define a tensor $\mathcal{X} \in \mathbb{R}^{2 \times 2 \times T}$ such that

$$x_{i,j,t} = \sum_{k=1}^t z_{i,j,k}$$

In this case, each mode-3 fiber of $\mathcal{X}$ follows a random walk. Both tensors can be generated and their mode-3 fibers corresponding to $i = j = 1$ can be plotted as follows:

```
scalar t = 100
scalar r = 2
scalar c = 2
scalar N = 3
```

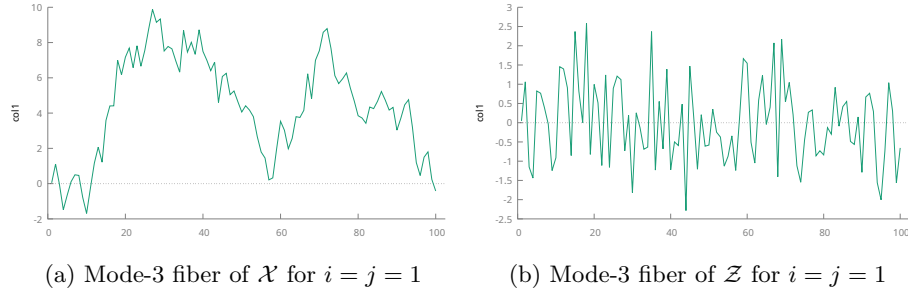


Figure 2

```

matrix mat = mnormal(t,r*c)
bundle X = tensorize(cum(mat),{t,r,c},1)
X = permute(X,{2,3,1})

# PLOT RANDOM WALK
bundle filts = defbundle("1",1,"2",1)
eval tensor_plot(X,3,filts)
# PLOT WHITE NOISE
Z = tensor_diff(X)
eval tensor_plot(Z,3,filts)

```

In the example above, the time dimension of $\mathcal{X}$ was initially set to be the first dimension by passing the scalar 1 as the third argument to **tensorize**. The tensor dimensions were then reordered using the **permute** function, which rearranges tensor dimensions and automatically updates the **time_id** key when present. Since time was moved to the last dimension, the **time_id** value was updated to 3. With the function **tensor_diff** $\mathcal{Z}$ was computed as:

$$\mathcal{Z}_t = \mathcal{X}_t - \mathcal{X}_{t-1}, \quad t = 2, \dots, T$$

Figure 2 displays the mode-3 fibers of $\mathcal{X}$ and $\mathcal{Z}$ for $i = j = 1$. The function **tensor_lag** follows exactly the same logic, but first order lags are computed instead of first differences:

```

bundle Xlag = tensor_lag(X,1,0)
eval X.tensor[1]

0.046320    -0.28566
-0.19392    -1.0960

eval Xlag.tensor[2]

0.046320    -0.28566
-0.19392    -1.0960

```

4.2 Tensor Autoregressive Models

Following Li and Xiao [2021] a tensor time-series $\{\mathcal{X}_t\}_{t=1}^T$ is a Tensor Autoregressive Process of order P (TenAR(P)), if:

$$\mathcal{X}_t = \sum_{p=1}^P \sum_{r=1}^{R_p} \mathcal{X}_{t-p} \times_1 \mathbf{A}_1^{(p,r)} \times_2 \dots \times_N \mathbf{A}_N^{(p,r)} + \mathcal{E}_t \quad (2)$$

Where $\mathcal{X}_t \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_N}$ represents an N -dimensional tensor. The square matrix $\mathbf{A}_k^{(p,r)} \in \mathbb{R}^{(d_k \times d_k)}$ contains the coefficients corresponding to the k -th dimension, p -th lag, and r -th rank². Finally, the tensor $\mathcal{E}_t$ is a tensor white noise satisfying $\text{Cov}(\text{vec}(\mathcal{E}_t), \text{vec}(\mathcal{E}_s)) = 0$ if $s \neq t$. However, the elements of $\mathcal{E}_t$ can be cross-sectionally correlated. For a complete description of TenAR(P), see Li and Xiao [2021].

If we fix the number of lags to one, 2 becomes:

$$\mathcal{X}_t = \sum_{r=1}^R \mathcal{X}_{t-1} \times_1 \mathbf{A}_1^{(r)} \times_2 \dots \times_N \mathbf{A}_N^{(r)} + \mathcal{E}_t$$

Applying the $\text{vec}()$ operator to the TenAR(1), we can rewrite it as a VAR(1):

$$\text{vec}(\mathcal{X}_t) = \left(\sum_{r=1}^R \mathbf{A}_N^{(r)} \otimes \mathbf{A}_{N-1}^{(r)} \otimes \dots \otimes \mathbf{A}_1^{(r)} \right) \text{vec}(\mathcal{X}_{t-1}) + \text{vec}(\mathcal{E}_t)$$

Li and Xiao [2021] call the scalar R the tensor-rank. A rank-one tensor of order K is the outer-product of K vectors. The CP rank of a tensor $\mathcal{X}$ is the minimum number of rank-1 tensors needed to express $\mathcal{X}$ as their sum [Kolda and Bader, 2009]. Given the properties of the vector outer-product ($\circ$), the following condition holds:

$$\begin{aligned} \Phi &= \mathbf{A}_N \otimes \mathbf{A}_{N-1} \otimes \dots \otimes \mathbf{A}_1 \\ \mathcal{R}(\Phi) &= \left(\text{vec}(\mathbf{A}_1) \circ \text{vec}(\mathbf{A}_2) \circ \dots \circ \text{vec}(\mathbf{A}_N) \right) \end{aligned}$$

where $\mathcal{R}$ is a permutation operator that matches the $(q \times q)$ matrix Φ , $q = \prod_{k=1}^N d_k$, to an order N tensor with dimensions $d_1 \times d_2 \times \dots \times d_N$. Given this property, the elements of the coefficient matrices of a TenAR(1) can be cast into an order N tensor:

$$\mathcal{R}(\Phi) = \sum_{r=1}^R \left(\text{vec}(\mathbf{A}_1^{(r)}) \circ \text{vec}(\mathbf{A}_2^{(r)}) \circ \dots \circ \text{vec}(\mathbf{A}_N^{(r)}) \right)$$

Hence Li and Xiao [2021] call R the tensor-rank, referring to the CP rank of $\mathcal{R}(\Phi)$. If we fix R to one, and we allow the number of lags to be greater than one, model (2) is:

$$\mathcal{X}_t = \sum_{p=1}^P \mathcal{X}_{t-p} \times_1 \mathbf{A}_1^{(p)} \times_2 \dots \times_N \mathbf{A}_N^{(p)} + \mathcal{E}_t$$

²The concept of rank in Li and Xiao [2021] will be detailed later in this section.

Applying the vec operator we have:

$$vec(\mathcal{X}_t) = \sum_{p=1}^P \Phi_p vec(\mathcal{X}_{t-p}) + vec(\mathcal{E}_t)$$

where

$$\Phi_p = \mathbf{A}_N^{(p)} \otimes \mathbf{A}_{N-1}^{(p)} \otimes \mathbf{A}_1^{(p)}, \quad p = 1, \dots, P$$

And of course this is a VAR(P) where coefficient matrices result from kronecker products of N smaller matrices, one per each dimension of the tensor excluding time. As such, the VAR(P) can be naturally rewritten as a VAR(1) adopting the companion matrix.

In practice, Li and Xiao [2021] propose two types of estimators for (2), one based on Least-Squares (LS), the other on Maximum-Likelihood (ML)³. In the **TenTS** package we consider the case where $R = 1, P \geq 1$ and provide ways to apply both the estimation methods through the function **tenar_est**. Note that if $N = 2$, the model coincides with a Matrix Autoregression [Chen et al., 2021]. In this case, the **MAR** package for **gretl** is recommended [Bucci et al., 2026]. In sections (4.2.1), (4.2.2) we describe the two estimation methods, and their implementation in our package. In the appendix we also show a comparison with results obtained with the **tensorTS** package for R.

4.2.1 LS Estimator

LS estimator for the TenAR(P) is given by:

$$(\hat{\mathbf{A}}_1^{(1)}, \dots, \hat{\mathbf{A}}_N^{(P)}) = \underset{(\hat{\mathbf{A}}_1^{(1)}, \dots, \hat{\mathbf{A}}_N^{(P)})}{argmin} \sum_t \left\| \mathcal{X}_t - \sum_{p=1}^P \mathcal{X}_{t-p} \times_1 \hat{\mathbf{A}}_1^{(p)} \times_2 \dots \times_N \hat{\mathbf{A}}_N^{(p)} \right\|_F^2 \quad (3)$$

However, there is not a closed form solution to find the minimizers of such a loss function. Li and Xiao [2021] adopt the Alternating Least Squares (ALS) algorithm [Kolda and Bader, 2009] to estimate the coefficient matrices. In the **TenTS** version of the ALS, coefficient matrices are all initialized as diagonal matrices with 0.5 in the diagonal. Then, given all the other matrices fixed, estimates for coefficient matrix of dimension k , lag i can be sequentially updated by the following expression:

$$\mathbf{A}_k^{(i)} = \sum_t \left[\left(\mathbf{X}_{t(k)} - \sum_{p \neq i} \mathbf{A}_k^{(p)} \mathbf{W}_{t-1, (k)}^{(p)} \right) \left(\mathbf{W}_{t-1, (k)}^{(i)} \right)' \right] \left[\mathbf{W}_{t-1, (k)}^{(i)} \left(\mathbf{W}_{t-1, (k)}^{(i)} \right)' \right]^{-1} \quad (4)$$

where

$$\Phi_k^{(p)} = \mathbf{A}_N^{(p)} \otimes \dots \otimes \mathbf{A}_{k+1}^{(p)} \otimes \mathbf{A}_{k-1}^{(p)} \otimes \dots \otimes \mathbf{A}_1^{(p)}; \quad \mathbf{W}_{t(k)}^{(p)} = \mathbf{X}_{t(k)} \left(\Phi_k^{(p)} \right)'$$

and $\mathbf{X}_{t(k)}$ is the matricization of the observed tensor in time t , dimension k .

To guarantee identifiability, the matrices $\mathbf{A}_k^{(i)}, k = 1, \dots, N-1$ are normalized

³We will occasionally use the terms LSE and MLE to denote the LS estimator and ML estimator, respectively.

such that they have frobenius norm of one. The procedure adopted in the **TenTS** package for LS estimation is described in algorithm (1):

Algorithm 1 LS Estimation and Normalization Procedure

Input: Current estimates of all coefficient matrices $\mathbf{A}_k^{(i)}$ for all dimensions $k = 1, \dots, N$ and lags $i = 1, \dots, P$.

Output: Updated and normalized coefficient matrices for all lags.

```

1: for  $i = 1$  to  $P$  do
2:    $\triangleright$  Update the Matrices
3:   for  $k = 1$  to  $N$  do
4:     Update  $\mathbf{A}_k^{(i)}$  with expression (4)
5:     if  $k < N$  then
6:       norm = frob. norm of  $\mathbf{A}_k^{(i)}$ 
7:        $\mathbf{A}_k^{(i)} = \mathbf{A}_k^{(i)} / \text{norm}$ 
8:     end if
9:   end for
10: end for

```

At each iteration, the difference between the updated matrices and the old values are computed. Define by $\mathbf{A}_{k,j}^{(i)}$ the values taken by coefficient matrix for dimension k , lag i at iteration j . At the end of each iteration, compute the norm of the differences:

$$\mathbf{d}_{k,j}^{(i)} = \left\| \mathbf{A}_{k,j}^{(i)} - \mathbf{A}_{k,j-1}^{(i)} \right\|_F^2, \quad i = 1, \dots, P, k = 1, \dots, N$$

Algorithm stops when the maximum of these norms is below a given tolerance, or the maximum number of iterations is reached. Once all coefficient matrices have been estimated, residuals can be computed as:

$$\hat{\mathcal{E}}_t = \mathcal{X}_t - \sum_{p=1}^P \mathcal{X}_{t-p} \times_1 \hat{\mathbf{A}}_1^{(p)} \times_2 \dots \times_N \hat{\mathbf{A}}_N^{(p)}$$

Then, an estimate of the covariance matrix for the idiosyncratic part is:

$$\hat{\Sigma}_{LSE} = \frac{1}{T-P} \sum_{t=P+1}^T \text{vec}(\hat{\mathcal{E}}_t) \text{vec}(\hat{\mathcal{E}}_t)'$$

Since the matrices $\hat{\mathbf{A}}_1^{(1)}, \dots, \hat{\mathbf{A}}_N^{(1)}$ contain a number of elements equal to $d = \sum_{k=1}^N d_k^2$, their asymptotic covariance matrix must be square ($dP \times dP$), symmetric, and positive definite. First, at each time t , call $q = \prod_{k=1}^N d_k$, and define the $(d \times q)$ matrix $\mathbf{W}_t^{(i)}$ by stacking the jacobian matrices of $\text{vec}(\mathcal{X}_t)$ with respect to the coefficient matrices for all N dimensions and lag i :

$$\mathbf{W}_t^{(i)} = \begin{bmatrix} \frac{\partial \text{vec}(\mathcal{X}_t)}{\partial \text{vec}(\mathbf{A}_1^{(i)})}, & \frac{\partial \text{vec}(\mathcal{X}_t)}{\partial \text{vec}(\mathbf{A}_2^{(i)})}, & \dots, & \frac{\partial \text{vec}(\mathcal{X}_t)}{\partial \text{vec}(\mathbf{A}_N^{(i)})} \end{bmatrix}' \quad (5)$$

That corresponds to:

$$\mathbf{W}_t^{(i)} = \begin{pmatrix} ((\mathbf{X}_{t-i,(1)} \Phi_1^{(i)'}) \otimes \mathbf{I}_{d_1}) \mathbf{Q}_1 \\ \vdots \\ ((\mathbf{X}_{t-i,(N)} \Phi_N^{(i)'}) \otimes \mathbf{I}_{d_N}) \mathbf{Q}_N \end{pmatrix},$$

where $\mathbf{Q}_k, k = 1, \dots, N$ are permutation matrices described in the appendix. Then, at each time period, vertically stack the jacobians that refer to all lags considered, obtaining a matrix with dimension $(Pd \times q)$:

$$\mathbf{W}_t = \begin{pmatrix} \mathbf{W}_t^1 \\ \vdots \\ \mathbf{W}_t^P \end{pmatrix}$$

Following section 4.1 of Li and Xiao [2021], the asymptotic covariance matrix for the coefficients estimator is given by:

$$\mathbf{\Xi}_{LSE} = \mathbf{H}^{-1} \mathbb{E} [\mathbf{W}_t \mathbf{\Sigma} \mathbf{W}_t'] \mathbf{H}^{-1}$$

where $\mathbf{\Sigma}$ can be estimated by $\hat{\mathbf{\Sigma}}_{LSE}$, and:

$$\mathbf{H} = \mathbb{E}(\mathbf{W}_t \mathbf{W}_t') + \sum_{i=1}^P \sum_{k=1}^{N-1} \gamma_k^i \gamma_k^{i'}$$

and $\mathbf{a}_k^{(i)} = \text{vec}(\mathbf{A}_k^{(i)})$, and $\gamma_k^{(i)} = (\mathbf{0}', \mathbf{a}_k^{(i)'}, \mathbf{0}')'$ is a $Pd \times 1$ vector, where the first $\mathbf{0}$ has $(d_1^2 + \dots + d_{k-1}^2) + d(i-1)$ rows, and the second $\mathbf{0}$ has a number of rows equal to $(d_{k+1}^2 + \dots + d_N^2) + d(P-i)$.

In practice, $\mathbf{H}$ is the expectation of the stacked jacobian matrix multiplied by its transpose and adjusted for elements of the coefficient matrices for lag $i, i = 1, \dots, P$ and dimension $k, k = 1, \dots, N-1$.

4.2.2 ML Estimator

As pointed out in Li and Xiao [2021], while for VAR(P) Least squares estimator coincide with the Maximum-Likelihood estimator under conditional gaussianity, this is not the case for higher order arrays. In fact, $\mathcal{E}_t$ following a tensor normal distribution implies the fact that its covariance matrix is separable, a condition that is not guaranteed by LSE. Separability means that the covariance of $\text{vec}(\mathcal{E}_t)$ is given by a limited number of kronecker products, one for each dimension of the tensor:

$$\text{cov}(\text{vec}(\mathcal{E}_t)) = \mathbf{\Sigma}_N \otimes \mathbf{\Sigma}_{N-1} \otimes \dots \otimes \mathbf{\Sigma}_1$$

If we assume that $\text{cov}(\text{vec}(\mathcal{E}_t))$ has this structure, ML estimator can be obtained through an alternating algorithm. Now, define the residual tensor after accounting for the P -lags:

$$\mathcal{R}_t = \mathcal{X}_t - \sum_{p=1}^P \mathcal{X}_{t-p} \times_1 \mathbf{A}_1^{(p)} \times_2 \dots \times_K \mathbf{A}_K^{(p)} \quad (6)$$

And the Partial Covariance Structure:

$$\mathbf{S}_k = \mathbf{\Sigma}_K \otimes \cdots \otimes \mathbf{\Sigma}_{k+1} \otimes \mathbf{\Sigma}_{k-1} \otimes \cdots \otimes \mathbf{\Sigma}_1$$

Then, the Log-Likelihood function is:

$$-\frac{1}{2}(T-P) \log 2\pi - \frac{1}{2} \sum_{k=1}^N \left[(T-P) \left(\prod_{l \neq k}^N d_l \right) \log |\Sigma_k| - \frac{1}{2} \sum_t \text{tr} \left[\Sigma_k^{-1} \mathbf{R}_{t(k)} \mathbf{S}_k^{-1} \mathbf{R}_{t(k)}' \right] \right]$$

where $\text{tr}(\cdot)$ and $|\cdot|$ respectively denote the trace and determinant operator. In practice, in MLE both the N covariance matrices and the NP coefficient matrices need to be initialized. For each lag, the **TenTS** package initializes the coefficient matrices as $(d_k \times d_k)$, $k = 1, \dots, N$ diagonal matrices, with 0.5 in the diagonal. On the other hand covariance matrices are initialized as identity matrices of dimensions $(d_k \times d_k)$, $k = 1, \dots, N$. Once an initial value is set for all of them, the gradient conditions determine the following iterative solution for the coefficient matrices:

$$\begin{aligned} \mathbf{A}_k^{(i)} = & \left[\sum_t \left(\mathbf{X}_{t(k)} - \sum_{p \neq i} \mathbf{A}_k^{(p)} \mathbf{X}_{t-p, (k)} \mathbf{\Phi}_k^{(p)'} \right) \mathbf{S}_k^{-1} \mathbf{\Phi}_k^{(i)} \mathbf{X}_{t-i, (k)}' \right] \times \\ & \times \left[\sum_t \mathbf{X}_{t-i, (k)} \mathbf{\Phi}_k^{(i)'} \mathbf{S}_k^{-1} \mathbf{\Phi}_k^{(i)} \mathbf{X}_{t-i, (k)}' \right]^{-1} \end{aligned} \quad (7)$$

Then, given the residuals obtained with the updated value of $\hat{\mathbf{A}}_k^{(i)}$, the covariance matrix for k -th dimension is updated in the following way:

$$\mathbf{\Sigma}_k = \frac{\sum_t \mathbf{R}_{t(k)} \mathbf{S}_k^{-1} \mathbf{R}_{t(k)}'}{(T-P) \prod_{l \neq k} d_l} \quad (8)$$

To identify the model, in this case we adopt a different normalization with respect to the LSE one, mainly to obtain results that are close to the R package **tensorTS** [Chen et al., 2022a]. In practice, for i -th lag, we update all coefficient matrices and covariances adopting the two expressions above. Then, once they are all updated, we divide coefficient matrix k , $k = 1, \dots, N-1$ by its largest singular value, and multiply coefficient matrix for dimension N for the product of the same $N-1$ singular values. Similarly, we divide first $N-1$ covariance matrices by their first eigenvalue, and multiply N -th covariance matrix for the product of these eigenvalues. Once this normalization phase is completed, we update matrices that refer to lag $i+1$. The procedure adopted in the **TenTS** package for ML estimation is described in algorithm (2):

Algorithm 2 ML Estimation and Normalization Procedure

Input: Current estimates of all coefficient matrices $\mathbf{A}_k^{(i)}$ for all dimensions $k = 1, \dots, N$ and lags $i = 1, \dots, P$, and all covariance matrices $\mathbf{\Sigma}_k$ for $k = 1, \dots, N$.

Output: Updated and normalized coefficient matrices for all lags. Updated and normalized covariance matrices.

```

1: for  $i = 1$  to  $P$  do

2:    $\triangleright$  Update the Coefficient Matrices

3:   for  $k = 1$  to  $N$  do
4:     Update  $\mathbf{A}_k^{(i)}$  with expression (7)
5:     Compute Residuals with expression (6)
6:     Update  $\mathbf{\Sigma}_k$  with expression (8)
7:   end for

8:    $\triangleright$  Normalize Coefficient Matrices using singular values

9:   prod_sing  $\leftarrow 1$ 
10:  for  $k = 1$  to  $N - 1$  do
11:     $\sigma_{\max} \leftarrow$  largest singular value of  $\mathbf{A}_k^{(i)}$ 
12:     $\mathbf{A}_k^{(i)} \leftarrow \mathbf{A}_k^{(i)} / \sigma_{\max}$ 
13:    prod_sing  $\leftarrow$  prod_sing  $\times \sigma_{\max}$ 
14:  end for
15:   $\mathbf{A}_N^{(i)} \leftarrow \mathbf{A}_N^{(i)} \times$  prod_sing

16:   $\triangleright$  Normalize Covariance Matrices using eigenvalues

17:  prod_eig  $\leftarrow 1$ 
18:  for  $k = 1$  to  $N - 1$  do
19:     $\lambda_1 \leftarrow$  first eigenvalue of  $\mathbf{\Sigma}_k$ 
20:     $\mathbf{\Sigma}_k \leftarrow \mathbf{\Sigma}_k / \lambda_1$ 
21:    prod_eig  $\leftarrow$  prod_eig  $\times \lambda_1$ 
22:  end for
23:   $\mathbf{\Sigma}_N \leftarrow \mathbf{\Sigma}_N \times$  prod_eig

24: end for
  
```

The algorithm utilizes the same stopping criterion as the LSE method. Upon termination, for each lag, the coefficient matrix of the final dimension is multiplied by the product of the norms of the other dimensions' coefficient matrices. Subsequently, the coefficient matrices for all lags and all dimensions—except the last one—are normalized to have a Frobenius norm of one. Given the assumption of separability, the estimate for the covariance matrix of the idiosyncratic part is:

$$\hat{\mathbf{\Sigma}}_{MLE} = \hat{\mathbf{\Sigma}}_N \otimes \hat{\mathbf{\Sigma}}_{N-1} \otimes \dots \otimes \hat{\mathbf{\Sigma}}_1$$

And the parameter covariance matrix for MLE is the following:

$$\Xi_{MLE} = \dot{\mathbf{H}}^{-1} \mathbb{E}(\mathbf{W}_t \Sigma^{-1} \mathbf{W}_t') \dot{\mathbf{H}}^{-1}$$

where

$$\dot{\mathbf{H}} = \mathbb{E}(\mathbf{W}_t \Sigma_{MLE}^{-1} \mathbf{W}_t) + \sum_{i=1}^P \sum_{k=1}^{N-1} \gamma_k^i \gamma_k^{i'}$$

and $\mathbf{W}_t, \gamma_k^i$ are defined in section (4.2.1).

5 Package Functions

5.1 Creating Tensors and Accessing Metadata

5.1.1 `tensorize`

Arguments: **matrix** `x`, **matrix** `dims`, **matrix** `time.id`(optional), **arrays** `dimnames`(optional)

Return type: **bundle**

Transforms a matrix into a tensor, where size of each dimension is specified by the matrix `dims`, which is assumed to be a row vector. Number of tensor dimensions is then given by number of columns of the matrix `dims`. The tensor is populated following column-order of the original matrix.

If the tensor order is 1 or 2, the function returns a bundle containing a matrix. If the tensor order is 3, the function returns a bundle containing an element of type matrices. If tensor order is greater than 3, the function returns a bundle containing an element of type array of arrays. In any case, they can be retrieved from the bundle by indexing it with the **tensor** bundle key. For example:

```
matrix x = seq(1,125)
bundle Y = tensorize(x,{5,5,5})
eval Y.tensor[1]
```

1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24
5	10	15	20	25

Note that number of elements in the input matrix must be equal to the product of all elements of `dims`.

The optional argument **time.id** is intended for labelling one of the tensor dimensions as the dimension related to time. This is important for time-series functions like **tensor.diff** and **tensor.lag**. Finally, the optional argument **dimnames** is for labelling each element of the N dimensions of the tensor with a name. Output of the function is then a bundle containing either two, three or four keys: the first one is **tensor**, the second one is **dims**, which contains sizes of the dimensions of the tensor. Last two keys are both optional and are **time.id** and **dimnames**.

5.1.2 get_tensor_dims

Arguments: **bundle** tensor

Return type: **matrix**

Returns the dimensions of a tensor stored in a row vector. Note that the argument of type bundle must contain an object of type matrix, matrices or array of arrays, which needs to be accessed through the key **tensor**. For example, the following script will produce an error:

```
matrices Z = msplitby(mshape(seq(1,20)',10,2),2)
bundle Y = _(elem_1=Z)
eval get_tensor_dims(Y)
```

While the correct version is the following:

```
matrices Z = msplitby(mshape(seq(1,20)',10,2),2)
bundle Y = _(tensor=Z)
eval get_tensor_dims(Y)
```

And the output is:

```
2  2  5
```

If the bundle was created by using **tensorize**, the problem will not arise. Also, if the bundle was created by using **tensorize**, dimensions of the tensor can be retrieved by accessing the keyword **dims** of the bundle.

5.2 Tensor Manipulation

5.2.1 tensor_3d

Arguments: **bundle** tensor

Return type: **matrices**

Transforms the tensor of order N , $N \geq 3$ into an object of type matrices, with resulting number of elements given by the product of dimensions from 3 to N . The bundle passed as argument must contain an object of type matrix, matrices or array of arrays under the key **tensor**. For example:

```
bundle Y = tensorize(seq(1,48),{2,4,3,2})
matrices Y_3d = tensor_3d(Y)
scalar n_matrices = nelem(Y_3d)
```

Where **n_matrices** is equal to 6.

5.2.2 vectorize_tensor

Arguments: **bundle** tensor

Return type: **matrix**

Transforms the tensor into a column vector, with number of rows equal to the product of the dimensions of the original tensor. The bundle passed as argument must contain an object of type matrix, matrices or array of arrays under the key **tensor**.

5.2.3 unfold

Arguments: **bundle** tensor, **scalar** n

Return type: **matrix**

Computes n-th mode of the tensor. The bundle passed as argument must contain an object of type matrix, matrices or array of arrays under the key **tensor**. The returned matrix has number of rows equal to the size of the n-th dimension, number of columns equal to the product of the sizes of all dimensions except for the n-th one. For example:

```
bundle Y = tensorize(seq(1,12),{2,2,3})
eval unfold(Y,1)
```

```
  1  3  5  7  9 11
  2  4  6  8 10 12
```

Note that if the object contained in the bundle is of type matrices, same results can be obtained more efficiently with the **flatten** function. In fact, the expression:

```
eval flatten(Y.tensor)
```

is equivalent to the one shown above.

5.2.4 permute

Arguments: **bundle** tensor, **matrix** new_order

Return type: **bundle**

Changes order of tensor dimensions, following order specified in argument **new_order**. For example, start by defining a tensor of order 4 in the following way:

```
bundle Y = tensorize(seq(1,400),{10,2,5,4})
matrix new_order = {1,2,4,3}
bundle X = permute(Y,new_order)
```

The following expression returns the initial matrix of dimensions:

```
eval get_tensor_dims(Y)
```

```
10 2 5 4
```

While in the second bundle order for the last two dimensions is inverted:

```
eval get_tensor_dims(X)
```

```
10 2 4 5
```

Note that number of columns of **new_order** must be equal to number of tensor dimensions. Also, the bundle passed as argument must contain an object of type matrix, matrices or array of arrays under the key **tensor**. Also, if the bundle keys **time_id** and **dimnames** exist, they will be automatically updated following the order specified in **new_order**. For an example see section 4.

5.2.5 tensor_drill

Arguments: **bundle** A, **bundle** filter_ids

Return type: **bundle**

Similar to the **drill** function in the **extra** package, this function extracts a subset of a tensor based on index selections provided by the **filter_ids**. The keys in **filter_ids** are the dimensions to filter, and they must be associated to a matrix containing indexes to keep along those dimensions.

In the example below, the code selects all elements of the tensor A whose second dimension index is either 2 or 3. Filters for additional dimensions can be applied in the same way, by supplying non-empty filter matrices for those dimensions:

```
matrix dims = {2,4,3,2}
matrix a = seq(1, prodr(dims))
A = tensorize(a, dims)
filt_A = tensor_drill(A, defbundle("2",{2,3}))
eval filt_A.dims

2 2 3 2
```

5.3 Tensor Statistics and Operations

5.3.1 tensor_mean

Arguments: **bundle** tensor, **scalar** index, **bool** squeeze[0]

Return type: **bundle**

Given an order N tensor, returns a tensor containing averages of tensor along dimension specified by **index**. The default output is a tensor of order N , where dimensions used to compute averages has now a size equal to 1. If the input **squeeze** is passed with a value of 1, the returned tensor has $N - 1$ dimensions. The bundle passed as argument must contain an object of type matrix, matrices or array of arrays under the key **tensor**.

5.3.2 tensor_aggr

Arguments: **bundle** tensor, **scalar** index, **bool** squeeze[0]

Return type: **bundle**

Similar to **tensor_mean**, given an order N tensor, returns a tensor containing sum of the tensor along dimension specified by **index**. The default output is a tensor of order N , where dimensions used to compute sum has now a size equal to 1. If the input **squeeze** is passed with a value of 1, the returned tensor has $N - 1$ dimensions. The bundle passed as argument must contain an object of type matrix, matrices or array of arrays under the key **tensor**.

5.3.3 tensor_sd

Arguments: **bundle** tensor, **scalar** index, **bool** squeeze[0]

Return type: **bundle**

Similar to **tensor_mean**, given an order N tensor, returns a tensor containing standard deviations of tensor along dimension specified by **index**. The default output is a tensor of order N , where dimensions used to compute standard deviations has now a size equal to 1. If the input **squeeze** is passed with a value of 1, the returned tensor has $N - 1$ dimensions. The bundle passed as argument must contain an object of type matrix, matrices or array of arrays under the key **tensor**.

5.3.4 **tensor_sum**

Arguments: **bundle A**, **bundle B**, **bool** which_name[0]

Return type: **bundle**

Computes the sum between the two tensors. The two bundles passed as arguments must contain an object of type matrix, matrices or array of arrays under the key **tensor**. Result is a tensor with same dimensions of the arguments, containing their sum. If the boolean **which_name** has a value of 1, resulting tensor will have same dimension names of the bundle B, otherwise it will have the dimension names of A.

5.3.5 **tensor_subtraction**

Arguments: **bundle A**, **bundle B**, **bool** which_name[0]

Return type: **bundle**

Computes the difference between the two tensors. The two bundles passed as arguments must contain an object of type matrix, matrices or array of arrays under the key **tensor**. Result is a tensor with same dimensions of the arguments, containing their difference. If the boolean **which_name** has a value of 1, resulting tensor will have same dimension names of the bundle B, otherwise it will have the dimension names of A.

5.3.6 **tensor_elementprod**

Arguments: **bundle A**, **bundle B**, **bool** which_name[0]

Return type: **bundle**

Computes the element-wise product between the two tensors. The two bundles passed as arguments must contain an object of type matrix, matrices or array of arrays under the key **tensor**. Result is a tensor with same dimensions of the arguments, containing their element-wise product. If the boolean **which_name** has a value of 1, resulting tensor will have same dimension names of the bundle B, otherwise it will have the dimension names of A.

5.3.7 **tensor_elementratio**

Arguments: **bundle A**, **bundle B**, **bool** which_name[0]

Return type: **bundle**

Computes the element-wise ratio between the two tensors. The two bundles passed as arguments must contain an object of type matrix, matrices or array

of arrays under the key **tensor**. Result is a tensor with same dimensions of the arguments, containing their element-wise ratio. If the boolean **which_name** has a value of 1, resulting tensor will have same dimension names of the bundle B, otherwise it will have the dimension names of A.

5.3.8 tensor_op

Arguments: **bundle** A, **bundle** B, **string** op, **bool** which_name[0]

Return type: **bundle**

Computes the element-wise operation specified by the string **op** between two tensors. The two bundles passed as arguments must contain an object of type matrix, matrices or array of arrays under the key **tensor**. For example:

```
matrix dims = {2,4,3}
a = seq(1,prodr(dims))
A = tensorize(a, dims)
Adouble = tensor_op(A,A,"+")
```

Where **Adouble** contains the sum of the bundle **A** with itself. Result is a tensor with same dimensions of the arguments. If the boolean **which_name** has a value of 1, resulting tensor will have same dimension names of the bundle B, otherwise it will have the dimension names of A.

5.3.9 tensor_outerprod

Arguments: **bundle** A, **bundle** B

Return type: **bundle**

Computes the outer product between the two tensors. The two bundles passed as arguments must contain an object of type matrix, matrices or array of arrays under the key **tensor**. Result is a tensor containing their outer product. For example:

```
matrix dims_a = {2,4,3}
a = seq(1,prodr(dims_a))
A = tensorize(a, dims_a)
dims_b = {4,3,2}
b = seq(1,prodr(dims_b))
B = tensorize(b, dims_b)
bundle ret = tensor_outerprod(A,B)
eval get_tensor_dims(ret)
```

And last line of the previous code will return the following:

```
2 4 3 4 3 2
```

5.3.10 tensor_lincomb

Arguments: **bundles** tensors, **matrix** coeffs

Return type: **bundle**

Computes the linear combination of the elements of the array of bundles in

tensors. Bundles are multiplied by the respective coefficient in **coeffs**, (which is assumed to be a vector containing one scalar per bundle), and then summed. The following example implements the difference between tensor A and B:

```

dims = {2,3,4}
bundle A = tensorize(mnormal(prodr(dims),1), dims)
bundle B = tensorize(mnormal(prodr(dims),1), dims)

bundles tensors = defarray(A,B)
matrix coeffs = {1, -1}
bundle C = tensor_lincomb(tensors,coeffs)

```

Where the same quantity could be obtained using **tensor_subtraction**.

5.3.11 **tensor_times_matrices**

Arguments: **bundle** tensor, **matrices** loads, **bool** transpose[0]
Return type: **bundle**

Computes the multilinear product between a tensor and an array of type matrices. The bundle passed as argument must contain an object of type matrix, matrices or array of arrays under the key **tensor**. The boolean **transpose** is zero by default, must be given a value of one if matrices in **loads** are to be transposed. If users want to compute the product on a subset of the dimensions of the tensors, (e.g. the n -mode product), they must define an empty array of type matrices with N elements, storing non-empty matrices on the right dimension index.

The returned tensor has the same number of dimensions of the one given as input. After multiplication, size of n -th dimension of resulting tensor will be equal to number of rows (columns if transpose is 1) of the respective loading matrix. The following example performs a multilinear product:

```

matrix dims = {2,3,4,5}
a = seq(1,prodr(dims))
B = tensorize(a, dims)
scalar N = cols(dims)
matrices loadings = array(N)
scalar r = 2

loop i = 1 .. N
    scalar c = dims[i]
    loadings[i] = mshape(seq(1,r*c),r,c)
endloop

bundle ret = tensor_times_matrices(B,loadings)
eval ret.dims
    2    2    2    2

```

Alternatively, here is an example for the n -mode product on the second dimension:

```

matrices loads = array(N)
loads[2] = mshape(seq(1,r*dims[2]),r,dims[2])
bundle ret = tensor_times_matrices(B,loads)
eval ret.dims
  2   2   4   5

```

Implementation of the product between tensor and element of type matrices is performed using equation 1 described in Kolda and Bader [2009], section 2.6.

5.4 Functions for Tensor-Valued Time Series

5.4.1 `tensor_lag`

Arguments: **bundle** A, **scalar** P[1], **bool** trim_zeros[1], **scalar** m[0]
Return type: **bundle**

Computes lags specified by the input P for a tensor time-series. In this case, the time dimension must be specified adding to the bundle A the key `time_id`. If the boolean `trim_zeros` has value 1, the first P observations along the time dimension will be removed from the tensor. Similar to the function `mlag`, if `trim_zeros` has a value of 0, the first P observations along the time dimension will have a value specified by the scalar `m`, the default being 0. For example:

```

matrix dims = {2,2,6}
A = tensorize(seq(1, prodr(dims)), dims,3)
Alag = tensor_lag(A)

eval unfold(Alag, A.time_id)

```

```

  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15  16
 17  18  19  20

```

And

```

Alag = tensor_lag(A,1,0)
eval unfold(Alag, A.time_id)

```

```

 0   0   0   0
 1   2   3   4
 5   6   7   8
 9  10  11  12
13  14  15  16
17  18  19  20

```

Where in this case the key `time_id` was directly specified in the `tensorize` function.

The bundle passed as argument must contain an object of type matrix, matrices or array of arrays under the key `tensor`.

5.4.2 `tensor_diff`

Arguments: **bundle** A, **scalar** P[1], **bool** trim_zeros[1]

Return type: **bundle**

Similar to `tensor_lag` computes difference between a tensor and its P-th lag computed on the time dimension. In this case, the time dimension must be specified adding to the bundle A the key `time_id`. If the boolean `trim_zeros` has a value of 1, first P observations in the time dimension are removed. If `trim_zeros` is equal to 0, first P observations in the time dimension are `nan`. The bundle passed as argument must contain an object of type matrix, matrices or array of arrays under the key `tensor`.

5.5 `tenar_est`

Arguments: **bundle** ten, **scalar** P[1], **bool** MLE[0], **bool** demean[0], **scalar** tol[0.00001], **scalar** max_iter[100], **bool** verbose[0]

Return type: **bundle**

Estimate a tensor autoregression from the tensor contained in the bundle `ten`. The user must specify which dimension refers to time through the key `time_id`. Also, the dimension that refers to time must be the last one, hence `time_id` must be equal to the last dimension of the tensor. If the number of tensor dimensions excluding time is 2, the model is equivalent to a Matrix Autoregression. In this case, the **MAR** package is recommended [Bucci et al., 2026]. The function prints the estimated coefficient matrices along with their standard errors, *z*-statistics, and corresponding *p*-values. The output is organized sequentially by lag: the function first prints the coefficient matrices corresponding to the first lag, ordered from dimension 1 to *N*. It then proceeds to print all subsequent coefficient matrices for lags 2, 3, ..., *P* in the same dimensional order.

5.5.1 Arguments of `tenar_est`

- **scalar** P: The number of lags considered, with a default value of 1.
- **bool** MLE: zero by default, hence the estimation method assumed by the function is the LSE (see 5.5). If a value of 1 is specified for **MLE**, the MLE estimator is adopted.
- **bool** demean: If 1, the sample mean is subtracted from the tensor before estimating the model. The mean is added back to estimated series.
- **scalar** tol: sets the tolerance of the algorithm.
- **scalar** max_iter: sets the maximum number of iterations of the algorithm.
- **bool** verbose: Argument to print the gain obtained in each iteration for the estimation of coefficients matrices. If this gain is less than **tol**, the algorithm stops. See section (4.2.1).

5.5.2 Outputs of `tenar_est`

- **bundle data**: bundle containing the tensor passed as input, from period $P + 1$ to T . This is meant to ease comparison between original data and estimates.
- **bundle estimates**: bundle containing values predicted by the model. They can be compared to the bundle **data** to understand the fitting of the model.
- **bundle residuals**: bundle containing the elementwise difference between the bundle **data** and the bundle **estimates**.
- **arrays param**: array of arrays containing estimated coefficients matrices. The number of elements of **param** is equal to P . Each element of **param** is an array containing N matrices, one per each dimension of the tensor excluding time.
- **arrays sd_param**: array of arrays containing standard errors for estimated coefficients matrices. The number of elements of **sd_param** is equal to P . Each element of **param** is an array containing N matrices, one per each dimension of the tensor excluding time.
- **matrix VCV**: Covariance matrix for the estimator. In practice, given P the selected number of lags, and d the sum of the square of the size of all dimensions excluding time, it is a $(Pd \times Pd)$ symmetric matrix containing covariances for the estimator of the coefficient matrices. Elements in **sd_param** are the diagonal elements of **VCV** arranged in a convenient way.
- **matrix Sigma**: Covariance matrix of the idiosyncratic part $vec(\mathcal{E}_t)$.
- **Others**: Scalars containing info about the model, such as the number of lags considered (**n_lags**), Log-likelihood (**llik**), information criteria (**AIC**, **BIC**, **HQC**) and the maximal eigenvalue of the companion matrix (**rho**).

5.6 Printouts and Plots

5.6.1 `tensor_print`

Arguments: **bundle** tensor

Return type: **void**

Prints all matrices of a tensor of order N , specifying respective dimensions from the third one to the $N - th$ one. The bundle passed as argument must contain an object of type matrix, matrices or array of arrays under the key **tensor**.

5.6.2 `tensor_plot`

Arguments: **bundle** A, **scalar** dim, **bundle** filt(optional), **matrix** aggr_mat(optional), ***matrix** time_mat(optional), **bool** do_means[1]

Return type: **void**

Plots the dimension of the tensor specified by index. Users can select a given subset of the total size of each dimension with the **filt** optional argument,

with the same syntax as **tensor_drill**. Users can also compute means along dimensions of the tensor, specifying the dimensions to compute means for in the **aggr_mat** matrix. If both **filt** and **aggr_mat** are passed as arguments, the function first extracts the specified subset of the tensor, and then computes the means. If the argument **do_means** is passed with a value of 0, the function computes the sum instead of the mean along dimensions specified in **aggr_mat**. The matrix containing plotted data can be retrieved with a pointer by the optional argument **time_mat**. The bundle passed as argument must contain an object of type matrix, matrices or array of arrays under the key **tensor**. Given an order-4 tensor, if one wants to plot its first dimension taking averages of the other three, one should do:

```
matrix dims = {6,3,4,2}
bundle X = tensorize(mnormal(dims[1],prodr(dims[-1])), dims)
matrix V = {}
eval tensor_plot(X,1,,{2,3,4}, &V)
```

5.7 Loading and Saving

5.7.1 store_tensor

Arguments: **bundle** A, **string** fname, **bool** export(optional)

Return type: **scalar**

Function to store the tensor. Follows exactly the same syntax of **bwrite**, that is the general function to store elements of type bundle.

5.7.2 read_tensor

Arguments: **string** fname, **bool** import(optional)

Return type: **scalar**

Function to read a tensor saved with either **tensor_store** or **bwrite**. Follows exactly the same syntax of **bread**, that is the general function to read elements of type bundle.

References

- A. Bucci, G. Palomba, and M. Tedeschi. Matrix-valued autoregressive (MAR) models in gretl. *Computational Statistics*, 41(3):47, 2026.
- R. Chen, H. Xiao, and D. Yang. Autoregressive models for matrix-valued time series. *Journal of Econometrics*, 222(1):539–560, 2021.
- R. Chen, Y. Han, Z. Li, H. Xiao, D. Yang, and R. Yu. Analysis of tensor time series: tensors. *Journal of Statistical Software*, 323:1–41, 2022a.
- R. Chen, D. Yang, and C.-H. Zhang. Factor models for high-dimensional tensor time series. *Journal of the American Statistical Association*, 117(537):94–116, 2022b.
- Y. Han, R. Chen, D. Yang, and C.-H. Zhang. Tensor factor model estimation by iterative projection. *The Annals of Statistics*, 52(6):2641–2667, 2024.
- T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.
- Z. Li and H. Xiao. Multi-linear tensor autoregressive models. *arXiv preprint arXiv:2110.00928*, 2021.

6 Appendix

6.1 Definition of Permutation Matrix

Permutation Matrix $\mathbf{P}_{m,n}$ is defined as:

$$\mathbf{P}_{m,n} = \sum_{i=1}^n \sum_{j=1}^m (\mathbf{U}_{ij} \otimes \mathbf{U}'_{ij})$$

where $\mathbf{U}_{ij}$ is the $n \times m$ matrix with one in (i, j) element and zeros elsewhere. Vector Permutation Matrix $\mathbf{Q}_k$ is defined as:

$$\mathbf{Q}_k = \mathbf{I}_{d_N d_{N-1} \dots d_{k+1}} \otimes \mathbf{P}_{d_k, d_{k-1} d_{k-2} \dots d_1}$$

where $\mathbf{I}$ is identity matrix and $\mathbf{P}$ is permutation matrix, $1 < k < N$. Especially, when $k = 1$, $\mathbf{Q}_1 = \mathbf{I}_{d_N d_{N-1} \dots d_1}$; when $k = N$, $\mathbf{Q}_N = \mathbf{P}_{d_N, d_{N-1} d_{N-2} \dots d_1}$

6.2 Comparison with R

In this section, we compare the results obtained with our package to those produced by the R package **tensorTS**. The analysis is conducted on monthly Fama–French portfolio returns formed on three firm characteristics: Operating Profitability, Book-to-Market Ratio, and Size.⁴ Operating Profitability and Book-to-Market Ratio have 4 levels, that go from low to high. Size has only two levels: Small and Big. We sample data from July 1963, to July 2025, resulting in

⁴The data are publicly available from the Kenneth R. French Data Library and are also provided in the **TentTS** package under the **examples** folder.

a tensor time-series of 745 months, where each tensor has dimensions $(4 \times 4 \times 2)$. Fixing the number of lags to one, we report the estimated coefficient matrices using the R package **tensorTS** for both LSE and MLE. For a comparison with **TenTS**, users can run the script **application.to.fama.french.inp** under the **examples** folder.

	1	2	3	4
1	0.06	-0.12	-0.04	0.49
2	-0.04	-0.07	0.05	0.37
3	-0.08	0.02	-0.04	0.43
4	-0.11	-0.16	0.05	0.60

Table 1: Coefficient matrix for Operating Profitability: LSE

	1	2	3	4
1	0.37	0.06	-0.41	0.17
2	0.28	0.07	-0.27	0.12
3	0.17	0.17	-0.27	0.14
4	0.26	0.24	-0.42	0.20

Table 2: Coefficient matrix for Book to Market: LSE

	1	2
1	1.47	0.36
2	0.63	0.34

Table 3: Coefficient matrix for Size: LSE

	1	2	3	4
1	0.30	0.22	0.05	-0.00
2	0.20	0.17	0.15	-0.07
3	0.23	-0.14	0.60	-0.13
4	0.04	0.44	0.28	-0.20

Table 4: Coefficient matrix for Operating Profitability: MLE

	1	2	3	4
1	0.19	0.09	-0.01	0.00
2	0.09	0.07	0.15	-0.03
3	0.01	0.10	0.10	0.08
4	-0.16	0.28	0.78	-0.43

Table 5: Coefficient matrix for Book to Market: MLE

	1	2
1	0.69	0.38
2	0.16	0.26

Table 6: Coefficient matrix for Size: MLE