



HAL
open science

Efficient algorithms to perform linear algebra operations on 3D arrays in vector languages

Francois Cuvelier

► **To cite this version:**

Francois Cuvelier. Efficient algorithms to perform linear algebra operations on 3D arrays in vector languages. 2018. hal-01809975

HAL Id: hal-01809975

<https://sorbonne-paris-nord.hal.science/hal-01809975>

Preprint submitted on 7 Jun 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Efficient algorithms to perform linear algebra operations on 3D arrays in vector languages

François Cuvelier*

2018/05/31

Abstract

In a few number of applications, a need arises to do some usual linear algebra operations on a very large number of very small matrices of the same size, referred in this report by 3D-array. These operations could be as simple as sum or products, or more complex like computation of determinants, factorizing, solving, ... The aim of this report is to describe some vectorized algorithms for each one of these operations and to give efficiency measures. For example, computing the LU decomposition with partial pivoting of one million of 8-by-8 matrices on our reference computer is performed in 3.1 seconds with Matlab, 5.6 seconds with Octave and 9.7 seconds with Python.

*Université Paris 13, Sorbonne Paris Cité, LAGA, CNRS UMR 7539, 99 Avenue J-B Clément, F-93430 Villetaneuse, France, cuvelier@math.univ-paris13.fr

This work was partially supported by the ANR project DEDALES under grant ANR-14-CE23-0005.

Contents

1	Notations and definitions	3
1.1	Element by element operations	4
1.2	Linear Algebra	5
1.2.1	Determinants	5
1.2.2	Matricial products	6
1.2.3	Linear systems	6
1.2.4	Positive Cholesky factorization	6
1.2.5	LU factorization with partial pivoting	6
2	Basic Linear Algebra Vectorized operations	7
2.1	Linear Combinations	7
2.2	Element by element operations	12
2.3	Matricial products	14
3	Linear solver for particular 3D-arrays	17
3.1	Diagonal matrices	17
3.2	Lower triangular matrices	18
3.3	Upper triangular matrices	19
4	Factorizations	25
4.1	Cholesky factorization	25
4.2	LU factorization with partial pivoting	28
4.2.1	Full computation	32
4.2.2	Inplace computation	36
5	Linear solvers	39
5.1	Using Cholesky factorization	39
5.2	Using LU factorization with partial pivoting	40
6	Determinants	44
6.1	Vectorized algorithm using the Laplace expansion	44
6.2	Using LU factorization	48
6.3	Vectorized algorithm using an other expansion	48
A	Vectorized algorithmic language	52
A.1	Common operators and functions	52
A.1.1	<code>SUB2IND</code> function	53
A.1.2	<code>IND2SUB</code> function	53
A.2	Combinatorial functions	53
B	Information for developpers	53

In this report we describe vectorized algorithms allowing some operations on a very large number of matrices of the same very small dimension: determinants, Cholesky or LU decomposition, solving ... These algorithms can be transposed in vectorized languages such as Matlab/Octave, Python, Scilab, Julia, ... provided that they contain 3D-arrays (or multidimensionnal arrays). The set of matrices is stored on a N -by- m -by- n 3D-array array where N is the number of matrices of dimensions m -by- n . In some vectorized languages, parts of these operations could be already implemented. They are referenced as *broadcasting* under Octave and Python, and as *expanding arrays with compatible sizes* under Matlab :

Matlab : https://fr.mathworks.com/help/matlab/matlab_prog/compatible-array-sizes-for-basic-operations.html

Octave : <https://www.gnu.org/software/octave/doc/v4.4.0/Broadcasting.html>

Python : <https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html>

In a first section, notations and definitions are given and some linear algebra operations such as linear combinations, matricial products, determinants, solving linear systems are extended to 3D-arrays. In the second section we introduce some algorithmic tools and functions used in this report. We use them on basic linear algebra operations on 3D-arrays to obtain various algorithms on linear combinations and matricial products. We also give very simple algorithms for elements by elements operations. Thereafter algorithms solving triangular upper systems and triangular lower systems stored in a 3D-array are described and their cputimes are computed for the three vectorized languages Matlab, Octave and Python. In section 4, various vectorized algorithms are detailed to obtain factorizations of all the matrices in a 3D-array: Cholesky factorization and LU factorization with partial pivoting are study. Then in section 5, some vectorized algorithms for solving linear systems stored in 3D-arrays are proposed. Finally, computation of determinants is study by using Laplace expansion or factorizations.

In all this report, for each kind of algorithms cputimes¹ is given for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

Some usual operations provided with vectorized languages are detailed in Appendix A.

All the source codes used can be

1 Notations and definitions

Some typographic conventions are used:

- \mathbb{Z} , \mathbb{N} , \mathbb{R} , \mathbb{C} are respectively the set of integers, positive integers, reals and complex numbers. \mathbb{K} is either \mathbb{R} or \mathbb{C} .
- All vectors or 1D-arrays are represented in bold : $\mathbf{v} \in \mathbb{R}^n$ or \mathbf{X} a 1D-array. The first alphabetic characters are **aAbBcC**

¹Computer with 1 Intel(R) Core(TM) i9-7940X CPU @ 3.10GHz processor (14 cores and 2 threads by core), 62GB of available RAM and Ubuntu 17.10 as operating system.

- All matrices or 2D-arrays are represented with the blackboard font as : $\mathbb{M} \in \mathcal{M}_{m,n}(\mathbb{K})$ or \mathbb{b} a m -by- n 2D-array. The first alphabetic characters are $\mathfrak{a}\mathbb{A}\mathfrak{b}\mathbb{B}\mathfrak{c}\mathbb{C}\dots$.
- All arrays of matrices or 3D-arrays are represented with the bold blackboard font as : $\mathbf{M} \in (\mathcal{M}_{m,n}(\mathbb{K}))^N$ or \mathbf{b} a m -by- n -by- p 3D-array. The first alphabetic characters are $\mathfrak{a}\mathbf{A}\mathfrak{b}\mathbf{B}\mathfrak{c}\mathbf{C}\dots$.

We now introduce some notations. Let $\mathbf{A} = (A_1, \dots, A_N) \in (\mathcal{M}_{m,n}(\mathbb{K}))^N$ where $N \gg m, n$. We identify \mathbf{A} with a N -by- m -by- n 3D-array such that

$$\mathbf{A}(k, :, :) = A_k, \quad \forall k \in \llbracket 1, N \rrbracket. \quad (1)$$

Thereafter, we said that a 3D-array $\mathbf{A} \in (\mathcal{M}_{m,n}(\mathbb{K}))^N$ has a matricial property if all the A_k matrices have this property. For example, \mathbf{A} is a symmetrical 3D-array if all its matrices are symmetrical.

In the following we described element by element operations on 3D-arrays.

1.1 Element by element operations

Some simple vectorized element by element operations (addition, subtraction, multiplication, ...) are often already coded with multidimensional arrays. For example the addition of two multidimensional arrays with same size or the addition of a multidimensional array with a scalar could be done with the $+$ operator with Matlab, Octave, Python, Scilab, ... Nevertheless, for our purpose we also want to be able to add a 3D-array with a 1D-array or with a matrix as described in the following subsection respectively in (5) and (3). These operations are not always available or are not directly usable on vectorized languages. For example with Matlab (\geq R2016b) or Octave

- to add a N -by- d -by- d 3D-array with a N -by-1 1D-array (see (5)) one can do

$$\text{randn}(N,d,d)+\text{randn}(N,1)$$

- to add a N -by- d -by- d 3D-array with a d -by- d 2D-array (see (3)) one cannot do $\text{randn}(N,d,d)+\text{randn}(d,d)$. The correct command is

$$\text{randn}(N,d,d)+\text{randn}(1,d,d) \text{ or } \text{randn}(N,d,d)+\text{reshape}(\text{randn}(d,d),[1,d,d])$$

We want to define the element by element operation

$$C \leftarrow A \diamond B$$

where A and/or B are 3D-arrays, \diamond is a vectorized element by element **binary** operator associated with the usual scalar operator \diamond . In Table 1, some examples of operators are provided.

Let $\mathbf{A} \in (\mathcal{M}_{m,n}(\mathbb{K}))^N$, We define four kinds of such operations:

1. Let $\mathbf{B} \in (\mathcal{M}_{m,n}(\mathbb{K}))^N$, we set

$$\mathbf{A} \diamond \mathbf{B} \stackrel{\text{def}}{=} \mathbf{C} \in (\mathcal{M}_{m,n}(\mathbb{K}))^N \quad (2)$$

where $\forall k \in \llbracket 1, N \rrbracket$

$$\forall i \in \llbracket 1, m \rrbracket, \forall j \in \llbracket 1, n \rrbracket, \quad C_k(i, j) = A_k(i, j) \diamond B_k(i, j).$$

Table 1: Common element by element operations

Operation	Algorithm			Matlab/Octave			Python		
	\diamond	\diamond	Name	Vec.	Sca.	Name	Vec.	Sca.	Name
multiply	.*	*	TIMES	.*	*	times	*	*	multiply
divide	./	/	RDIVIDE	./	/	rdivide	/	/	divide
add	+	+	PLUS	+	+	plus	+	+	add
subtract	-	-	MINUS	-	-	minus	-	-	subtract
power	.^	^	POWER	.^	^	power	**	**	power
divide	.\	\	LDIVIDE	.\	\	ldivide			

2. Let $\mathbb{B} \in \mathcal{M}_{m,n}(\mathbb{K})$, we set

$$\mathbf{A} \diamond \mathbb{B} \stackrel{\text{def}}{=} \mathbf{C} \in (\mathcal{M}_{m,n}(\mathbb{K}))^N \quad (3)$$

$$\mathbb{B} \diamond \mathbf{A} \stackrel{\text{def}}{=} \mathbf{D} \in (\mathcal{M}_{m,n}(\mathbb{K}))^N \quad (4)$$

where $\forall k \in \llbracket 1, N \rrbracket$

$$\forall i \in \llbracket 1, m \rrbracket, \forall j \in \llbracket 1, n \rrbracket, \begin{cases} \mathbb{C}_k(i, j) = \mathbb{A}_k(i, j) \diamond \mathbb{B}(i, j), \\ \mathbb{D}_k(i, j) = \mathbb{B}(i, j) \diamond \mathbb{A}_k(i, j). \end{cases}$$

3. Let $\mathbf{B} \in \mathbb{K}^N$, we set

$$\mathbf{A} \diamond \mathbf{B} \stackrel{\text{def}}{=} \mathbf{C} \in (\mathcal{M}_{m,n}(\mathbb{K}))^N \quad (5)$$

$$\mathbf{B} \diamond \mathbf{A} \stackrel{\text{def}}{=} \mathbf{D} \in (\mathcal{M}_{m,n}(\mathbb{K}))^N \quad (6)$$

where $\forall k \in \llbracket 1, N \rrbracket$

$$\forall i \in \llbracket 1, m \rrbracket, \forall j \in \llbracket 1, n \rrbracket, \begin{cases} \mathbb{C}_k(i, j) = \mathbb{A}_k(i, j) \diamond \mathbf{B}(k), \\ \mathbb{D}_k(i, j) = \mathbf{B}(k) \diamond \mathbb{A}_k(i, j). \end{cases}$$

4. Let $B \in \mathbb{K}$, we set

$$\mathbf{A} \diamond B \stackrel{\text{def}}{=} \mathbf{C} \in (\mathcal{M}_{m,n}(\mathbb{K}))^N \quad (7)$$

$$B \diamond \mathbf{A} \stackrel{\text{def}}{=} \mathbf{D} \in (\mathcal{M}_{m,n}(\mathbb{K}))^N \quad (8)$$

where $\forall k \in \llbracket 1, N \rrbracket$

$$\forall i \in \llbracket 1, m \rrbracket, \forall j \in \llbracket 1, n \rrbracket, \begin{cases} \mathbb{C}_k(i, j) = \mathbb{A}_k(i, j) \diamond B, \\ \mathbb{D}_k(i, j) = B \diamond \mathbb{A}_k(i, j). \end{cases}$$

1.2 Linear Algebra

1.2.1 Determinants

Let $\mathbf{A} \in (\mathcal{M}_{n,n}(\mathbb{K}))^N$. The determinant of \mathbf{A} , denoted by $\det \mathbf{A}$, is the vector $\mathbf{D} \in \mathbb{K}^N$ such that

$$\mathbf{D}(k) = \det(k, :, :), \quad \forall k \in \llbracket 1, N \rrbracket. \quad (9)$$

1.2.2 Matricial products

Let X be in $(\mathcal{M}_{m,n}(\mathbb{K}))^N$ or $\mathcal{M}_{m,n}(\mathbb{K})$, and Y be in $(\mathcal{M}_{n,p}(\mathbb{K}))^N$ or $\mathcal{M}_{n,p}(\mathbb{K})$, where either one of the two is a 3D-array. We extend the matricial product to 3D-arrays

$$X * Y = \mathbf{Z} \in (\mathcal{M}_{m,p}(\mathbb{K}))^N \quad (10)$$

where $\forall k \in \llbracket 1, N \rrbracket$

$$\mathbf{Z}(k, :, :) = \begin{cases} X(k, :, :) * Y(k, :, :), & \text{if } X \in (\mathcal{M}_{m,n}(\mathbb{K}))^N \text{ and } Y \in (\mathcal{M}_{n,p}(\mathbb{K}))^N, \\ X(k, :, :) * Y, & \text{if } X \in (\mathcal{M}_{m,n}(\mathbb{K}))^N \text{ and } Y \in \mathcal{M}_{n,p}(\mathbb{K}), \\ X * Y(k, :, :), & \text{if } X \in \mathcal{M}_{m,n}(\mathbb{K}) \text{ and } Y \in (\mathcal{M}_{n,p}(\mathbb{K}))^N. \end{cases}$$

In these formulas, the operator $*$ denotes the matricial product between a m -by- n matrix and a n -by- p matrix.

1.2.3 Linear systems

Let $\mathbf{A} \in (\mathcal{M}_{n,n}(\mathbb{K}))^N$ and let B be in $(\mathcal{M}_{n,p}(\mathbb{K}))^N$ or in $\mathcal{M}_{n,p}(\mathbb{K})$. We want to find $\mathbf{X} \in (\mathcal{M}_{n,p}(\mathbb{K}))^N$ such that

$$\mathbf{A} * \mathbf{X} = B \quad (11)$$

that is to say, for all $k \in \llbracket 1, N \rrbracket$, find $\mathbb{X}_k \in \mathcal{M}_{d,n}(\mathbb{K})$ solution of the linear system

$$\mathbb{A}_k * \mathbb{X}_k = \begin{cases} B_k, & \text{if } B \in (\mathcal{M}_{n,p}(\mathbb{K}))^N, \\ B, & \text{if } B \in \mathcal{M}_{n,p}(\mathbb{K}). \end{cases}$$

1.2.4 Positive Cholesky factorization

Let $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{C}))^N$ be a hermitian positive definite 3D-array:

$$\forall k \in \llbracket 1, N \rrbracket, \mathbf{A}(k, :, :) = \mathbb{A}_k \in \mathcal{M}_n(\mathbb{C}) \text{ is a hermitian positive definite matrix.}$$

The positive Cholesky factorization of \mathbf{A} is given by

$$\mathbf{A} = \mathbf{L}\mathbf{L}^* \quad (12)$$

where $\mathbf{L} \in (\mathcal{M}_{d,d}(\mathbb{C}))^N$ is lower triangular with real and positive diagonal entries (i.e. $\forall k \in \llbracket 1, N \rrbracket$, $\mathbf{L}(k, :, :) = \mathbb{L}_k \in \mathcal{M}_n(\mathbb{C})$ are lower triangular matrices with real and positive diagonal entries) The equation (12) can be equivalently written as

$$\forall k \in \llbracket 1, N \rrbracket, \mathbb{A}_k = \mathbb{L}_k \mathbb{L}_k^*.$$

1.2.5 LU factorization with partial pivoting

Let $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$. The LU factorization with partial pivoting of \mathbf{A} is given by

$$\mathbf{P}\mathbf{A} = \mathbf{L}\mathbf{U} \quad (13)$$

where \mathbf{L} , \mathbf{U} and \mathbf{P} are in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$ and

- \mathbf{L} is a lower triangular 3D-array with unit diagonal, i.e. $\forall k \in \llbracket 1, N \rrbracket$, $\mathbf{L}(k, :, :) = \mathbb{L}_k \in \mathcal{M}_d(\mathbb{K})$ are lower triangular matrices with unit diagonal,

- \mathbf{U} is a upper triangular 3D-array,
i.e. $\forall k \in \llbracket 1, N \rrbracket$, $\mathbf{U}(k, :, :) = \mathbb{U}_k \in \mathcal{M}_d(\mathbb{K})$ are upper triangular matrices,
- \mathbf{P} is a permutation 3D-array,
i.e. $\forall k \in \llbracket 1, N \rrbracket$, $\mathbf{P}(k, :, :) = \mathbb{P}_k \in \mathcal{M}_d(\mathbb{K})$ are permutation matrices,

So we have, for all $k \in \llbracket 1, N \rrbracket$,

$$\mathbb{P}_k \mathbb{A}_k = \mathbb{L}_k \mathbb{U}_k$$

2 Basic Linear Algebra Vectorized operations

To introduce some algorithmic functions we present, in the next subsection, various versions of the linear combinations function called `AXPBXY` :

- `AXPBXY_CPT` , component by component computation (3 loops)
- `AXPBXY_MAT` , using 2D-array or matricial operations (1 loop over number of matrices),
- `AXPBXY_VEC` , vectorized algorithm (2 loops over rows and columns of the matrices),
- `AXPBXY_CVT` , using operations between 3D-arrays (no loop)

Thereafter same versions are written for for element by elements functions. At last, theses versions are provided for the matricial product function called `MTIMES` .

2.1 Linear Combinations

Let X and Y be in $(\mathcal{M}_{m,n}(\mathbb{K}))^N$, $\mathcal{M}_{m,n}(\mathbb{K})$, \mathbb{K}^N or \mathbb{K} where either one of the two is in $(\mathcal{M}_{m,n}(\mathbb{K}))^N$. Let α and β in \mathbb{K} , we can compute

$$\mathbf{Z} = \alpha X + \beta Y \in (\mathcal{M}_{m,n}(\mathbb{K}))^N \quad (14)$$

by using one of the formulas (2) to (8). In Python with Numpy, Matlab ($\geq 2016b$) and Octave ($\geq 4.0.3?$) such operations are partially supported. Let $X \in (\mathcal{M}_{m,n}(\mathbb{K}))^N$, $Y \in \mathcal{M}_{m,n}(\mathbb{K})$ and $Z \in \mathbb{K}^N$, we want to compute

$$C = 7 * X - 5 * Y \text{ and } D = 3 * X + 4 * Z$$

- with Matlab ($\geq 2016b$) and Octave, one has to expand Y to a 1-by- m -by- n 3D-array to fit broadcasting rules by using `reshape` function:

```
N=10^5;m=3;n=2;
X=randn(N,m,n);Y=randn(m,n);Z=randn(N,1);
C=7*X-5*reshape(Y,[1,m,n]);
D=3*X+4*Z;
```


- with Matlab (<2016b), one has to expand Y to a N -by- m -by- n 3D-array by using `reshape` and `repmat` functions:

```
N=10^5;m=3;n=2;
X=randn(N,m,n);Y=randn(m,n);Z=randn(N,1);
C=7*X-5*repmat(reshape(Y,[1,m,n]),N,1,1);
D=3*X+4*repmat(Z,1,m,n);
```

- with Python, one has to expand Z to a N -by-1-by-1 3D-array to fit broadcasting rules by using `reshape` function:

```
import numpy as np
N=10^5;m=3;n=2
X=np.random.randn(N,m,n)
Y=np.random.randn(m,n)
Z=np.random.randn(N)
C=7*X-5*Y
D=3*X+4*np.reshape(Z,[N,1,1])
```

To introduce the functions `GETCPT`, `GETMAT`, `GETVEC` and `TO3DARRAY` we now present some algorithms implementing (14) without using broadcasting.

A very basic function called `AXPBY_CPT` using three loops is given in Algorithm 1 where computation of \mathbf{Z} is done component by component.

Algorithm 1 Function `AXPBY_CPT`, returns linear combination $\alpha X + \beta Y$ by using component by component computation.

```
Function  $\mathbf{Z} \leftarrow \text{AXPBY\_CPT}(\alpha, X, \beta, Y)$ 
  for  $k \leftarrow 1$  to  $N$  do
    for  $i \leftarrow 1$  to  $m$  do
      for  $j \leftarrow 1$  to  $n$  do
         $\mathbf{Z}(k, i, j) \leftarrow \alpha * \text{GETCPT}(X, k, i, j)$ 
           $+ \beta * \text{GETCPT}(Y, k, i, j)$ 
      end for
    end for
  end for
end Function
```

Algorithm 2 Function `GETCPT`, returns component (i, j) of the k -th matrix of X .

Input X : in $(\mathcal{M}_{m,n}(\mathbb{K}))^N$ or $\mathcal{M}_{m,n}(\mathbb{K})$
or in \mathbb{K}^N or in \mathbb{K} ,
 k : matrix index,
 i : row index,
 j : column index

Output s : a scalar.

```
Function  $s \leftarrow \text{GETCPT}(X, k, i, j)$ 
  if  $X \in \mathbb{K}$  then
     $s \leftarrow X$ 
  else if  $X \in \mathbb{K}^N$  then
     $s \leftarrow X(k)$ 
  else if  $X \in \mathcal{M}_{m,n}(\mathbb{K})$  then
     $s \leftarrow X(i, j)$ 
  else
     $s \leftarrow X(k, i, j)$   $\triangleright X \in (\mathcal{M}_{m,n}(\mathbb{K}))^N$ 
  end if
end Function
```

We present in Algorithm 3 an other version where linear combination of two multidimensional arrays with same size supposed to be in our vectorized language: this version is quite efficient but memory consuming

Algorithm 3 Function `AXPBY_CVT`, returns linear combination $\alpha X + \beta Y$ by converting arrays to a 3D-arrays.

```

Function Z  $\leftarrow$  AXPBY_CVT ( $\alpha, X, \beta, Y$ )
  Z  $\leftarrow$   $\alpha * \text{TO3DARRAY}(X, N, m, n)$ 
            $+\beta * \text{TO3DARRAY}(Y, N, m, n)$ 
end Function

```

Algorithm 4 Function `TO3DARRAY`, converts to a 3Darray

```

Input  X : in  $(\mathcal{M}_{m,n}(\mathbb{K}))^N$  or  $\mathcal{M}_{m,n}(\mathbb{K})$ 
         or in  $\mathbb{K}^N$  or in  $\mathbb{K}$ ,
        N : number of  $m$ -by- $n$  matrices,
        m : number of rows,
        n : number of columns,
Output T : in  $(\mathcal{M}_{m,n}(\mathbb{K}))^N$ .

```

```

Function T  $\leftarrow$  TO3DARRAY ( $X, N, m, n$ )
if X in  $\mathbb{K}$ . then
  T  $\leftarrow X * \text{ONES}(N, m, n)$ 
else if  $X \in \mathbb{K}^N$  then
  T  $\leftarrow \text{REPTILE}(X, 1, m, n)$ 
else if  $X \in \mathcal{M}_{m,n}(\mathbb{K})$  then
  T  $\leftarrow \text{REPTILE}(X, N, 1, 1)$ 
else
  T  $\leftarrow X$   $\triangleright X \in (\mathcal{M}_{m,n}(\mathbb{K}))^N$ 
end if
end Function

```

An other way is to use operations defined on 2D-array (or matrices) which are supposed to be defined in the vectorized language: that's give the Algorithm 5.

Algorithm 5 Function `AXPBY_MAT`, returns linear combination $\alpha X + \beta Y$ by using vectorized operations on 2D-arrays or matrices.

```

Function Z  $\leftarrow$  AXPBY_MAT ( $\alpha, X, \beta, Y$ )
  for k  $\leftarrow$  1 to N do
    Z(k, :, :)  $\leftarrow$   $\alpha * \text{GETMAT}(X, k)$ 
                    $+\beta * \text{GETMAT}(Y, k)$ 
  end for
end Function

```

Algorithm 6 Function `GETMAT`, returns the k -th matrix of X .

```

Input  X : in  $(\mathcal{M}_{m,n}(\mathbb{K}))^N$  or  $\mathcal{M}_{m,n}(\mathbb{K})$ 
         or in  $\mathbb{K}^N$  or in  $\mathbb{K}$ ,
        k : matrix index,
Output M : in  $\mathbb{K}$  or in  $\mathcal{M}_{m,n}(\mathbb{K})$ .

```

```

Function M  $\leftarrow$  GETMAT ( $X, k$ )
if  $X \in \mathbb{K}$  or  $X \in \mathcal{M}_{m,n}(\mathbb{K})$  then
  M  $\leftarrow X$ 
else if  $X \in \mathbb{K}^N$  then
  M  $\leftarrow X(k)$ 
else
  M  $\leftarrow X(k, :, :)$   $\triangleright X \in (\mathcal{M}_{m,n}(\mathbb{K}))^N$ 
end if
end Function

```

As N supposed to be very large in front of n and m , the Algorithm 5 is not efficient: the main loop to suppress is the loop over N . This is the object of the Algorithm 7.

Algorithm 7 Function `AXPBY_VEC`, returns linear combination $\alpha X + \beta Y$ by using vectorized operations on 1D-arrays.

```

Function Z  $\leftarrow$  AXPBY_VEC ( $\alpha, X, \beta, Y$ )
  for i  $\leftarrow$  1 to m do
    for j  $\leftarrow$  1 to n do
      Z(:, i, j)  $\leftarrow$   $\alpha * \text{GETVEC}(X, i, j)$ 
                     $+\beta * \text{GETVEC}(Y, i, j)$ 
    end for
  end for
end Function

```

Algorithm 8 Function `GETVEC`, returns (i, j) components of X .

```

Input  X : in  $(\mathcal{M}_{m,n}(\mathbb{K}))^N$  or  $\mathcal{M}_{m,n}(\mathbb{K})$ 
         or in  $\mathbb{K}^N$  or in  $\mathbb{K}$ ,
        i : row index,
        j : column index
Output V : in  $\mathbb{K}^N$  or in  $\mathbb{K}$ .

```

```

Function V  $\leftarrow$  GETVEC ( $X, i, j$ )
if  $X \in \mathbb{K}^N$  or in  $\mathbb{K}$ . then
  V  $\leftarrow X$ 
else if  $X \in \mathcal{M}_{m,n}(\mathbb{K})$  then
  V  $\leftarrow X(i, j)$ 
else
  V  $\leftarrow X(:, i, j)$   $\triangleright X \in (\mathcal{M}_{m,n}(\mathbb{K}))^N$ 
end if
end Function

```

In Table 2, the computation time in second of these four functions under Matlab, Octave and Python are given when the input arrays are $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$ and with a number N up to 10^5 for the slower function `AXPBY_CPT` and up to 10^6 for the others. As expected the two functions `AXPBY_VEC` and `AXPBY_CVT` are the fastest. Indeed when broadcasting is available in vector language these two functions are less efficient than the broadcasting one given in Table 2 by the function `AXPBY_NAT`. In Table 3 and 4 computation time in second of the `AXPBY_NAT` function and for N up to 10^7 are given when the input array $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and the input array Y respectively in $\mathcal{M}_{3,3}(\mathbb{R})$ and in $(\mathcal{M}_{3,3}(\mathbb{K}))^N$. In Tables 5 to 8 effects of multi-threading with Matlab are provided respectively for the functions `AXPBY_MAT`, `AXPBY_VEC`, `AXPBY_CVT` and `AXPBY_NAT`.

N	Matlab	Octave	Python	N	Matlab	Octave	Python
20 000	0.025(s)	4.829(s)	0.176(s)	200 000	0.013(s)	0.014(s)	0.007(s)
40 000	0.039(s)	9.683(s)	0.351(s)	400 000	0.017(s)	0.026(s)	0.018(s)
60 000	0.057(s)	14.520(s)	0.537(s)	600 000	0.030(s)	0.089(s)	0.049(s)
80 000	0.078(s)	19.439(s)	0.717(s)	800 000	0.038(s)	0.118(s)	0.065(s)
100 000	0.096(s)	24.237(s)	0.899(s)	1 000 000	0.045(s)	0.146(s)	0.082(s)

(a) Function <code>AXPBY_CPT</code>				(b) Function <code>AXPBY_CVT</code>			
N	Matlab	Octave	Python	N	Matlab	Octave	Python
200 000	0.382(s)	5.658(s)	0.361(s)	200 000	0.008(s)	0.009(s)	0.015(s)
400 000	0.760(s)	11.349(s)	0.723(s)	400 000	0.011(s)	0.014(s)	0.033(s)
600 000	1.138(s)	17.118(s)	1.094(s)	600 000	0.020(s)	0.029(s)	0.059(s)
800 000	1.544(s)	22.823(s)	1.459(s)	800 000	0.032(s)	0.039(s)	0.081(s)
1 000 000	1.894(s)	28.534(s)	1.812(s)	1 000 000	0.040(s)	0.048(s)	0.102(s)

(c) Function <code>AXPBY_MAT</code>				(d) Function <code>AXPBY_VEC</code>			
N	Matlab	Octave	Python	N	Matlab	Octave	Python
200 000	0.006(s)	0.005(s)	0.004(s)	200 000	0.006(s)	0.005(s)	0.004(s)
400 000	0.006(s)	0.009(s)	0.008(s)	400 000	0.006(s)	0.009(s)	0.008(s)
600 000	0.011(s)	0.034(s)	0.033(s)	600 000	0.011(s)	0.034(s)	0.033(s)
800 000	0.019(s)	0.045(s)	0.044(s)	800 000	0.019(s)	0.045(s)	0.044(s)
1 000 000	0.022(s)	0.055(s)	0.054(s)	1 000 000	0.022(s)	0.055(s)	0.054(s)

(e) Function <code>AXPBY_NAT</code>			
N	Matlab	Octave	Python
200 000	0.006(s)	0.005(s)	0.004(s)
400 000	0.006(s)	0.009(s)	0.008(s)
600 000	0.011(s)	0.034(s)	0.033(s)
800 000	0.019(s)	0.045(s)	0.044(s)
1 000 000	0.022(s)	0.055(s)	0.054(s)

Table 2: Computational times in seconds of `AXPBY` functions with $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

N	Matlab	Matlab(*)	Octave	Python
200 000	0.006(s)	0.009(s)	0.006(s)	0.004(s)
400 000	0.007(s)	0.012(s)	0.011(s)	0.009(s)
600 000	0.016(s)	0.035(s)	0.024(s)	0.022(s)
800 000	0.019(s)	0.045(s)	0.045(s)	0.044(s)
1 000 000	0.022(s)	0.056(s)	0.055(s)	0.055(s)
5 000 000	0.065(s)	0.268(s)	0.268(s)	0.273(s)
10 000 000	0.120(s)	0.521(s)	0.526(s)	0.545(s)

Table 3: Computational times in seconds of `AXPBY_NAT` functions with X in $(\mathcal{M}_{3,3}(\mathbb{K}))^N$ and Y in $\mathcal{M}_{3,3}(\mathbb{R})$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. Matlab(*) refers to Matlab without multi-threadings.

N	Matlab	Matlab(*)	Octave	Python
200 000	0.003(s)	0.008(s)	0.007(s)	0.005(s)
400 000	0.004(s)	0.014(s)	0.016(s)	0.016(s)
600 000	0.015(s)	0.050(s)	0.050(s)	0.037(s)
800 000	0.019(s)	0.066(s)	0.067(s)	0.049(s)
1 000 000	0.024(s)	0.082(s)	0.083(s)	0.062(s)
5 000 000	0.092(s)	0.408(s)	0.415(s)	0.311(s)
10 000 000	0.171(s)	0.793(s)	0.805(s)	0.608(s)

Table 4: Computational times in seconds of `AXPBY_NAT` functions with X and Y in $(\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. Matlab(*) refers to Matlab without multi-threadings.

N	1 threads	2 threads	4 threads	6 threads	8 threads	14 threads	20 threads	28 threads
200 000	0.382(s)	0.393(s)	0.386(s)	0.382(s)	0.376(s)	0.376(s)	0.376(s)	0.377(s)
400 000	0.763(s)	0.762(s)	0.761(s)	0.755(s)	0.751(s)	0.750(s)	0.750(s)	0.762(s)
600 000	1.136(s)	1.134(s)	1.141(s)	1.146(s)	1.168(s)	1.143(s)	1.134(s)	1.143(s)
800 000	1.517(s)	1.525(s)	1.514(s)	1.511(s)	1.510(s)	1.502(s)	1.511(s)	1.518(s)
1 000 000	1.921(s)	1.897(s)	1.897(s)	1.886(s)	1.890(s)	1.901(s)	1.899(s)	1.910(s)
5 000 000	9.582(s)	9.494(s)	9.458(s)	9.545(s)	9.516(s)	9.566(s)	9.514(s)	9.533(s)
10 000 000	19.406(s)	19.468(s)	19.383(s)	19.638(s)	19.480(s)	19.598(s)	19.705(s)	19.991(s)

Table 5: Function `AXPBY_MAT` with $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$ under Matlab 2018a: effects of multithreading on cputimes

N	1 threads	2 threads	4 threads	6 threads	8 threads	14 threads	20 threads	28 threads
200 000	0.008(s)	0.005(s)	0.006(s)	0.006(s)	0.007(s)	0.006(s)	0.007(s)	0.006(s)
400 000	0.014(s)	0.011(s)	0.011(s)	0.011(s)	0.013(s)	0.012(s)	0.013(s)	0.012(s)
600 000	0.024(s)	0.024(s)	0.025(s)	0.025(s)	0.025(s)	0.026(s)	0.026(s)	0.026(s)
800 000	0.032(s)	0.032(s)	0.032(s)	0.033(s)	0.033(s)	0.033(s)	0.033(s)	0.033(s)
1 000 000	0.040(s)	0.039(s)	0.040(s)	0.039(s)	0.040(s)	0.040(s)	0.040(s)	0.040(s)
5 000 000	0.307(s)	0.292(s)	0.281(s)	0.277(s)	0.276(s)	0.276(s)	0.274(s)	0.274(s)
10 000 000	0.617(s)	0.584(s)	0.561(s)	0.554(s)	0.552(s)	0.557(s)	0.549(s)	0.551(s)

Table 6: Function `AXPBY_VEC` with $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$ under Matlab 2018a: effects of multithreading on cputimes

N	1 threads	2 threads	4 threads	6 threads	8 threads	14 threads	20 threads	28 threads
200 000	0.018(s)	0.009(s)	0.012(s)	0.012(s)	0.012(s)	0.012(s)	0.012(s)	0.012(s)
400 000	0.030(s)	0.020(s)	0.017(s)	0.016(s)	0.016(s)	0.016(s)	0.017(s)	0.017(s)
600 000	0.047(s)	0.041(s)	0.034(s)	0.033(s)	0.032(s)	0.031(s)	0.031(s)	0.031(s)
800 000	0.077(s)	0.042(s)	0.041(s)	0.034(s)	0.038(s)	0.033(s)	0.039(s)	0.033(s)
1 000 000	0.096(s)	0.063(s)	0.047(s)	0.048(s)	0.045(s)	0.045(s)	0.045(s)	0.046(s)
5 000 000	0.474(s)	0.299(s)	0.177(s)	0.161(s)	0.153(s)	0.149(s)	0.148(s)	0.149(s)
10 000 000	0.935(s)	0.595(s)	0.334(s)	0.299(s)	0.278(s)	0.273(s)	0.263(s)	0.265(s)

Table 7: Function `AXPBY_CVT` with $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$ under Matlab 2018a: effects of multithreading on cputimes

N	1 threads	2 threads	4 threads	6 threads	8 threads	14 threads	20 threads	28 threads
200 000	0.010(s)	0.006(s)	0.005(s)	0.005(s)	0.005(s)	0.005(s)	0.005(s)	0.005(s)
400 000	0.015(s)	0.008(s)	0.007(s)	0.006(s)	0.006(s)	0.006(s)	0.006(s)	0.006(s)
600 000	0.037(s)	0.023(s)	0.019(s)	0.018(s)	0.017(s)	0.016(s)	0.016(s)	0.016(s)
800 000	0.047(s)	0.030(s)	0.022(s)	0.021(s)	0.020(s)	0.021(s)	0.020(s)	0.020(s)
1 000 000	0.057(s)	0.035(s)	0.025(s)	0.025(s)	0.023(s)	0.022(s)	0.022(s)	0.023(s)
5 000 000	0.273(s)	0.163(s)	0.102(s)	0.082(s)	0.075(s)	0.065(s)	0.063(s)	0.061(s)
10 000 000	0.534(s)	0.310(s)	0.193(s)	0.153(s)	0.134(s)	0.122(s)	0.117(s)	0.110(s)

Table 8: Function `AXPBY_NAT` with $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$ under Matlab 2018a: effects of multithreading on cputimes

2.2 Element by element operations

From the four Algorithms 1, 3, 5, 7 and by using notations and definitions of section 1.1, we deduce four generic functions which computes $X \diamond Y$. There are given in Algorithms 9 to 11.

Algorithm 9 Function `EBYE_CPT`, returns element by element operation $X \diamond Y$. Here f is the function $f : (x, y) \in \mathbb{K}^2 \rightarrow x \diamond y$

```

Function Z ← EBYE_CPT ( $X, Y, f$ )
  for  $k \leftarrow 1$  to  $N$  do
    for  $i \leftarrow 1$  to  $m$  do
      for  $j \leftarrow 1$  to  $n$  do
         $x \leftarrow \text{GETCPT}(X, k, i, j)$ 
         $y \leftarrow \text{GETCPT}(Y, k, i, j)$ 
         $\mathbf{Z}(k, i, j) \leftarrow f(x, y)$ 
      end for
    end for
  end for
end Function

```

Algorithm 10 Function `EBYE_MAT`, returns element by element operation $X \diamond Y$ by using function $f : (\mathbb{A}, \mathbb{B}) \rightarrow \mathbb{A} \diamond \mathbb{B}$ where \mathbb{A} and \mathbb{B} are in $\mathcal{M}_{m,n}(\mathbb{K})$ or in \mathbb{K} .

```

Function C ← EBYE_MAT ( $X, Y, f$ )
  for  $k \leftarrow 1$  to  $N$  do
     $\mathbb{A} \leftarrow \text{GETMAT}(X, k)$ 
     $\mathbb{B} \leftarrow \text{GETMAT}(Y, k)$ 
     $\mathbf{C}(k, :, :) \leftarrow f(\mathbb{A}, \mathbb{B})$ 
  end for
end Function

```

Algorithm 11 Function `EBYE_VEC`, returns element by element operation $X \diamond Y$ by using function $f : (\mathbb{A}, \mathbb{B}) \rightarrow \mathbb{A} \diamond \mathbb{B}$ where \mathbb{A} and \mathbb{B} are in \mathbb{K}^N .

```

Function C ← EBYE_VEC ( $X, Y, f$ )
  for  $i \leftarrow 1$  to  $m$  do
    for  $j \leftarrow 1$  to  $n$  do
       $\mathbb{A} \leftarrow \text{GETVEC}(X, i, j)$ 
       $\mathbb{B} \leftarrow \text{GETVEC}(Y, i, j)$ 
       $\mathbf{C}(:, i, j) \leftarrow f(\mathbb{A}, \mathbb{B})$ 
    end for
  end for
end Function

```

Algorithm 12 Function `EBYE_CVT`, returns element by element operation $X \diamond Y$ by converting arrays to a 3D-arrays. Here f is the function $f : (\mathbb{A}, \mathbb{B}) \rightarrow \mathbb{A} \diamond \mathbb{B}$ where \mathbb{A} and \mathbb{B} are in $(\mathcal{M}_{m,n}(\mathbb{K}))^N$.

```

Function Z ← EBYE_CVT ( $X, Y, f$ )
   $\mathbb{A} \leftarrow \text{TO3DARRAY}(X, N, m, n)$ 
   $\mathbb{B} \leftarrow \text{TO3DARRAY}(Y, N, m, n)$ 
   $\mathbf{Z} \leftarrow f(\mathbb{A}, \mathbb{B})$ 
end Function

```

Thereafter, writing functions for a specific element by element operator is easy. For example, the corresponding functions for element by element multiplication

operator $\mathbf{.*}$ are provided in Algorithms 13 to 15.

Algorithm 13 Function `TIMES_CPT`, returns element by element product $X \mathbf{.*} Y$

```

Function  $\mathbf{Z} \leftarrow \text{TIMES\_CPT}(X, Y)$ 
   $f : (x, y) \rightarrow x * y$ 
   $\mathbf{Z} \leftarrow \text{EBYE\_CPT}(X, Y, f)$ 
end Function

```

Algorithm 14 Function `TIMES_MAT`, returns element by element product $X \mathbf{.*} Y$ by using vectorized operations on 2D-arrays or matrices.

```

Function  $\mathbf{Z} \leftarrow \text{TIMES\_MAT}(X, Y)$ 
   $f : (\mathbf{A}, \mathbf{B}) \rightarrow \mathbf{A} \mathbf{.*} \mathbf{B}$ 
   $\mathbf{Z} \leftarrow \text{EBYE\_MAT}(X, Y, f)$ 
end Function

```

Algorithm 15 Function `TIMES_VEC`, returns element by element product $X \mathbf{.*} Y$ by using vectorized operations on 1D-arrays.

```

Function  $\mathbf{Z} \leftarrow \text{TIMES\_VEC}(X, Y)$ 
   $f : (\mathbf{A}, \mathbf{B}) \rightarrow \mathbf{A} \mathbf{.*} \mathbf{B}$ 
   $\mathbf{Z} \leftarrow \text{EBYE\_MAT}(X, Y, f)$ 
end Function

```

Algorithm 16 Function `TIMES_CVT`, returns element by element product $X \mathbf{.*} Y$ by converting arrays to a 3D-arrays.

```

Function  $\mathbf{Z} \leftarrow \text{TIMES\_CVT}(X, Y)$ 
   $f : (\mathbf{A}, \mathbf{B}) \rightarrow \mathbf{A} \mathbf{.*} \mathbf{B}$ 
   $\mathbf{Z} \leftarrow \text{EBYE\_CVT}(X, Y, f)$ 
end Function

```

In Table 9, the computation time in second of these four functions under Matlab, Octave and Python are given when the input arrays are $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$. For the `TIMES_CPT` function the N values are up to 10^5 and for the other functions up to 10^6 . As expected the two functions `TIMES_VEC` and `TIMES_CVT` are the fastest. Indeed when broadcasting is available in vector language these two functions are less efficient than the broadcasting one given in Table 9 by the function `TIMES_NAT`.

N	Matlab	Octave	Python	N	Matlab	Octave	Python
200 000	0.660(s)	51.287(s)	3.741(s)	200 000	0.009(s)	0.013(s)	0.004(s)
400 000	1.336(s)	103.601(s)	7.495(s)	400 000	0.017(s)	0.026(s)	0.012(s)
600 000	2.012(s)	155.196(s)	11.262(s)	600 000	0.031(s)	0.057(s)	0.030(s)
				800 000	0.038(s)	0.076(s)	0.040(s)
				1 000 000	0.044(s)	0.093(s)	0.051(s)

(a) Function `TIMES_CPT`

N	Matlab	Octave	Python	N	Matlab	Octave	Python
200 000	0.471(s)	5.845(s)	0.180(s)	200 000	0.007(s)	0.008(s)	0.015(s)
400 000	0.951(s)	11.713(s)	0.361(s)	400 000	0.011(s)	0.012(s)	0.034(s)
600 000	1.395(s)	17.665(s)	0.550(s)	600 000	0.023(s)	0.026(s)	0.055(s)
800 000	1.899(s)	23.458(s)	0.742(s)	800 000	0.030(s)	0.034(s)	0.080(s)
1 000 000	2.325(s)	29.365(s)	0.921(s)	1 000 000	0.038(s)	0.041(s)	0.101(s)

(b) Function `TIMES_CVT`

N	Matlab	Octave	Python
200 000	0.004(s)	0.004(s)	0.002(s)
400 000	0.005(s)	0.005(s)	0.005(s)
600 000	0.009(s)	0.018(s)	0.012(s)
800 000	0.011(s)	0.023(s)	0.024(s)
1 000 000	0.013(s)	0.029(s)	0.030(s)

(c) Function `TIMES_MAT`

N	Matlab	Octave	Python
200 000	0.004(s)	0.004(s)	0.002(s)
400 000	0.005(s)	0.005(s)	0.005(s)
600 000	0.009(s)	0.018(s)	0.012(s)
800 000	0.011(s)	0.023(s)	0.024(s)
1 000 000	0.013(s)	0.029(s)	0.030(s)

(d) Function `TIMES_VEC`

(e) Function `TIMES_NAT`

Table 9: Computational times in seconds of `TIMES` functions with $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

2.3 Matricial products

In section 1.2.2 matricial product with 3D-arrays is defined and the corresponding function is called `MTIMES`.

From the three Algorithms 1, 7 and 5 we deduce three functions which computes (10). They are given respectively in Algorithms 17, 18 and 19.

Algorithm 17 Function `MTIMES_CPT`, returns matricial products $X * Y$ where X or/and Y are 3D-arrays.

```

Function C ← MTIMES_CPT ( $X, Y$ )
  for  $k \leftarrow 1$  to  $N$  do
    for  $i \leftarrow 1$  to  $m$  do
      for  $j \leftarrow 1$  to  $p$  do
         $S \leftarrow 0$ 
        for  $l \leftarrow 1$  to  $n$  do
           $S \leftarrow S +$ 
            GETCPT( $X, k, i, l$ ) * GETCPT( $Y, k, l, j$ )
        end for
        C( $k, i, j$ ) ←  $S$ 
      end for
    end for
  end for
end Function

```

Algorithm 18 Function `MTIMES_VEC`, returns matricial products $X * Y$ where X or/and Y are 3D-arrays.

```

Function C ← MTIMES_VEC ( $X, Y$ )
  for  $i \leftarrow 1$  to  $m$  do
    for  $j \leftarrow 1$  to  $p$  do
       $S \leftarrow \text{ZEROS}(N, 1)$ 
      for  $l \leftarrow 1$  to  $n$  do
         $S \leftarrow S + \text{GETVEC}(X, i, l) * \text{GETVEC}(Y, l, j)$ 
      end for
      C( $:, i, j$ ) ←  $S$ 
    end for
  end for
end Function

```

Algorithm 19 Function `MTIMES_MAT`, returns matricial products $X * Y$ where X or/and Y are 3D-arrays.

```

Function C ← MTIMES_MAT ( $X, Y$ )
  for  $k \leftarrow 1$  to  $N$  do
    C( $k, :, :$ ) ← GETMAT( $X, k$ ) * GETMAT( $Y, k$ )
  end for
end Function

```

In Python, the `matmul` function of the Numpy package ($\geq 1.10.0$) directly do these kind of operations and it's also implements the semantics of the `@` operator introduced in Python 3.5. In Matlab 2018a and Octave 4.4.0 no function is available to perform such operations.

In Table 10, computational time in second of theses three functions under Matlab, Octave and Python are given when the input arrays are $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$. For the `MTIMES_CPT` function the N value is up to 10^5 and for the other functions up to 10^6 . As expected the function `MTIMES_VEC` is the fastest.

In Table 11, computationnal time in second of the `MTIMES_VEC` function for the same input datas is given for N up to 10^7 . To see effect of (automatic) multithreading under Matlab, we also add computational times of the function when using only one thread. One can also compare with the native function in Python which uses the `matmul` function of the Numpy package.

In Table 12, computationnal time of the same functions are given but for X and Y in $(\mathcal{M}_{3,3}(\mathbb{K}))^N$.

N	Matlab	Octave	Python
200 000	0.587(s)	161.459(s)	5.146(s)
400 000	1.198(s)	325.489(s)	10.345(s)
600 000	1.809(s)	488.050(s)	15.453(s)
800 000	2.415(s)	650.824(s)	20.728(s)
1 000 000	3.015(s)	814.027(s)	25.739(s)

(a) Function `MTIMES_CPT`

N	Matlab	Octave	Python	N	Matlab	Octave	Python
200 000	0.471(s)	5.845(s)	0.180(s)	200 000	0.007(s)	0.008(s)	0.015(s)
400 000	0.951(s)	11.713(s)	0.361(s)	400 000	0.011(s)	0.012(s)	0.034(s)
600 000	1.395(s)	17.665(s)	0.550(s)	600 000	0.023(s)	0.026(s)	0.055(s)
800 000	1.899(s)	23.458(s)	0.742(s)	800 000	0.030(s)	0.034(s)	0.080(s)
1 000 000	2.325(s)	29.365(s)	0.921(s)	1 000 000	0.038(s)	0.041(s)	0.101(s)

(b) Function `MTIMES_MAT`(c) Function `MTIMES_VEC`Table 10: Computational times in seconds of `MTIMES` functions with $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

N	Matlab	Matlab(*)	Octave	Python	Python(Nat)
200 000	0.025(s)	0.021(s)	0.021(s)	0.031(s)	0.009(s)
400 000	0.046(s)	0.046(s)	0.038(s)	0.074(s)	0.017(s)
600 000	0.075(s)	0.071(s)	0.060(s)	0.120(s)	0.029(s)
800 000	0.100(s)	0.098(s)	0.081(s)	0.170(s)	0.046(s)
1 000 000	0.126(s)	0.184(s)	0.103(s)	0.220(s)	0.058(s)
5 000 000	1.070(s)	1.195(s)	0.776(s)	1.519(s)	0.292(s)
10 000 000	2.109(s)	2.355(s)	1.989(s)	3.006(s)	0.577(s)

Table 11: Computational times in seconds of `MTIMES_VEC` functions with X in $(\mathcal{M}_{3,3}(\mathbb{K}))^N$ and Y in $\mathcal{M}_{3,3}(\mathbb{R})$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. Matlab(*) refers to Matlab without multi-threadings and Python(Nat) to Numpy `matmul` function.

N	Matlab	Matlab(*)	Octave	Python	Python(Nat)
200 000	0.036(s)	0.028(s)	0.022(s)	0.051(s)	0.078(s)
400 000	0.088(s)	0.070(s)	0.043(s)	0.117(s)	0.159(s)
600 000	0.098(s)	0.099(s)	0.069(s)	0.182(s)	0.242(s)
800 000	0.173(s)	0.139(s)	0.093(s)	0.256(s)	0.327(s)
1 000 000	0.264(s)	0.181(s)	0.119(s)	0.322(s)	0.405(s)
5 000 000	1.516(s)	1.678(s)	1.079(s)	1.689(s)	2.032(s)
10 000 000	2.961(s)	3.290(s)	2.073(s)	3.697(s)	4.065(s)

Table 12: Computational times in seconds of `MTIMES_VEC` functions with X and Y in $(\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. Matlab(*) refers to Matlab without multi-threadings and Python(Nat) to Numpy `matmul` function.

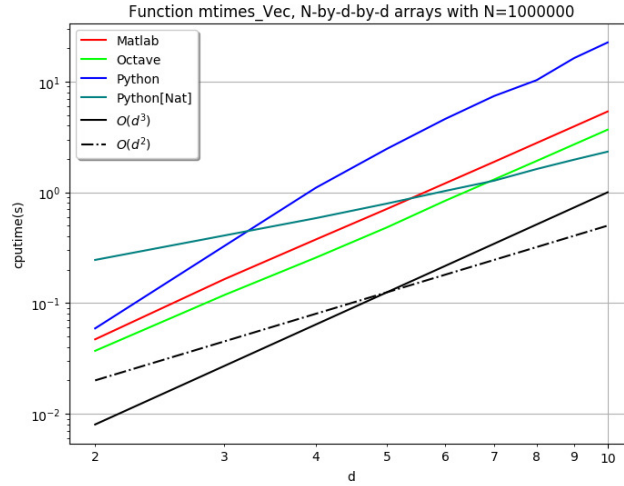


Figure 1: Computational times in seconds of the `MTIMES_VEC` function with \mathbf{X} and \mathbf{Y} both in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$, $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

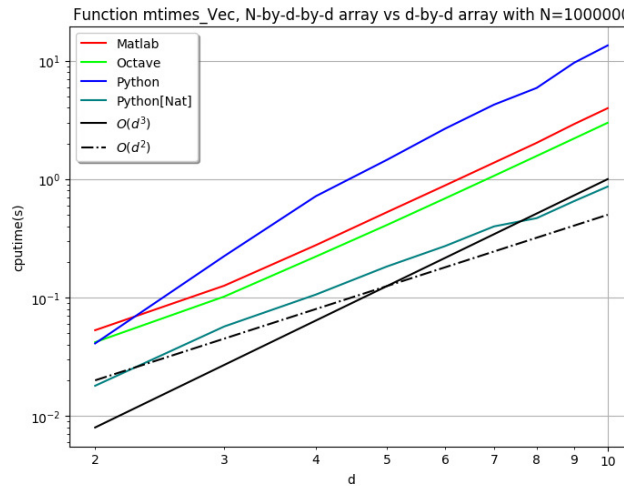


Figure 2: Computational times in seconds of the `MTIMES_VEC` function with $\mathbf{X} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$, $\mathbf{Y} \in \mathcal{M}_{d,d}(\mathbb{R})$, $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

3 Linear solver for particular 3D-arrays

In this section we suppose that $\mathbf{A} \in (\mathcal{M}_{n,n}(\mathbb{K}))^N$ is **regular** i.e. all the \mathbb{A}_k matrices are regular. Let B be in $(\mathcal{M}_{n,p}(\mathbb{K}))^N$ or in $\mathcal{M}_{n,p}(\mathbb{K})$. As defined in section 1.2.3, we want to find $\mathbf{X} \in (\mathcal{M}_{n,p}(\mathbb{K}))^N$ such that

$$\mathbf{A} * \mathbf{X} = B$$

In this section we give the **LINSOLVEDIAG** function and various versions of the **LINSOLVETRIU** and **LINSOLVETRIU** functions which solve linear systems respectively with \mathbf{A} diagonal, lower triangular and upper triangular.

3.1 Diagonal matrices

Firstly we recall some very simple results. Let $\mathbb{D} \in \mathcal{M}_{d,d}(\mathbb{K})$ be a regular diagonal matrix. If $\mathbf{b} \in \mathbb{K}^d$ then the solution $\mathbf{x} \in \mathbb{K}^d$ of the linear system $\mathbb{D}\mathbf{x} = \mathbf{b}$ is given by

$$\mathbf{x}_i = \mathbf{b}_i / \mathbb{D}_{i,i}, \quad \forall i \in \llbracket 1, d \rrbracket.$$

If $\mathbb{B} \in \mathcal{M}_{d,n}(\mathbb{K})$ then the solution $\mathbf{X} \in \mathcal{M}_{d,n}(\mathbb{K})$ of $\mathbb{D}\mathbf{X} = \mathbb{B}$ is given by

$$\mathbf{X}_{i,l} = \mathbb{B}_{i,l} / \mathbb{D}_{i,i}, \quad \forall i \in \llbracket 1, d \rrbracket, \forall l \in \llbracket 1, n \rrbracket.$$

Now, one can easily extend these results to **regular diagonal 3D-array**. Let $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$ be a **regular diagonal 3D-array**, i.e. each $\mathbb{A}_k \stackrel{\text{def}}{=} \mathbf{A}(k, :, :)$ is a regular diagonal matrix, and so $\forall k \in \llbracket 1, N \rrbracket, \forall (i, j) \in \llbracket 1, d \rrbracket^2$

$$\begin{aligned} \mathbb{A}_k(i, j) &= 0, \quad \text{if } i \neq j \\ \mathbb{A}_k(i, i) &\neq 0. \end{aligned}$$

Let $B \in (\mathcal{M}_{d,n}(\mathbb{K}))^N$ or $B \in \mathcal{M}_{d,n}(\mathbb{K})$, we want to solve the linear systems

$$\mathbf{A}\mathbf{X} = B$$

as described in section 1.2.3. If $B \in \mathcal{M}_{d,n}(\mathbb{K})$ then we have

$$\mathbf{X}(k, i, l) = B(i, l) / \mathbf{A}(k, i, i), \quad \forall i \in \llbracket 1, d \rrbracket, \forall l \in \llbracket 1, n \rrbracket, \forall k \in \llbracket 1, N \rrbracket$$

and if $B \in (\mathcal{M}_{n,m}(\mathbb{K}))^N$ we obtain

$$\mathbf{X}(k, i, l) = B(k, i, l) / \mathbf{A}(k, i, i), \quad \forall i \in \llbracket 1, d \rrbracket, \forall l \in \llbracket 1, n \rrbracket, \forall k \in \llbracket 1, N \rrbracket.$$

By using the function **GETVEC** described in Algorithm 8, we obtain the vectorized Algorithm 20.

Algorithm 20 Function `LINSOLVEDIAG` , solves diagonal linear system $\mathbb{A}\mathbb{X} = \mathbb{B}$.

Input \mathbf{A} : in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$
 \mathbf{B} : in $(\mathcal{M}_{d,n}(\mathbb{K}))^N$, or in $\mathcal{M}_{d,n}(\mathbb{K})$
Output \mathbf{X} : in $(\mathcal{M}_{d,n}(\mathbb{K}))^N$

Function $\mathbf{X} \leftarrow \text{LINSOLVEDIAG}(\mathbf{A}, \mathbf{B})$
 for $l \leftarrow 1$ **to** m **do**
 for $i \leftarrow 1$ **to** d **do**
 $\mathbb{X}(:, i, l) \leftarrow \text{GETVEC}(B, i, l) ./ \mathbf{A}(:, i, i)$
 end for
 end for
end Function

3.2 Lower triangular matrices

Firstly we recall some classical formulas. Let $\mathbb{A} \in \mathcal{M}_{d,d}(\mathbb{K})$ be a regular lower triangular matrix. If $\mathbb{B} \in \mathcal{M}_{d,n}(\mathbb{K})$ then the solution $\mathbb{X} \in \mathcal{M}_{d,n}(\mathbb{K})$ of

$$\mathbb{A}\mathbb{X} = \mathbb{B} \tag{15}$$

can be computed column by column. For each column l , we successively compute $\mathbb{X}_{1,l}, \mathbb{X}_{2,l}, \dots, \mathbb{X}_{d,l}$ by using formula

$$\mathbb{X}_{i,l} = (\mathbb{B}_{i,l} - \sum_{j=1}^{i-1} \mathbb{A}_{i,j} \mathbb{X}_{j,l}) / \mathbb{A}_{i,i}, \quad \forall i \in \llbracket 1, d \rrbracket, \quad \forall l \in \llbracket 1, n \rrbracket. \tag{16}$$

or in a more compact form we successively compute $\mathbb{X}_{1,:}, \mathbb{X}_{2,:}, \dots, \mathbb{X}_{d,:}$ by using formula

$$\mathbb{X}_{i,:} = (\mathbb{B}_{i,:} - \mathbb{A}_{i,1:i-1} \mathbb{X}_{1:i-1,:}) / \mathbb{A}_{i,i}, \quad \forall i \in \llbracket 1, d \rrbracket. \tag{17}$$

A such operation is given by the `LINSOLVETRIL` function described in Algorithm 21.

Now, one can extend these results to **regular lower triangular** 3D-array. Let $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$ be a **regular lower triangular** 3D-array, i.e. each $\mathbb{A}_k \stackrel{\text{def}}{=} \mathbf{A}(k, :, :)$ is a regular lower triangular matrix, and so $\forall k \in \llbracket 1, N \rrbracket, \forall (i, j) \in \llbracket 1, d \rrbracket^2$

$$\begin{aligned} \mathbb{A}_k(i, j) &= 0, & \text{if } i < j \\ \mathbb{A}_k(i, i) &\neq 0. \end{aligned}$$

By using `LINSOLVETRIL` and `GETMAT` functions respectively described in Algorithm 21 and Algorithm 6, we easily obtain the non-vectorized function `LINSOLVETRIL_MAT` written in Algorithm 22. In Algorithm 23, an other code is presented without using function `LINSOLVETRIL_MAT` . This code uses `GETCPT` function given in Algorithm 2 and by permuting the main loop in k with the two others in l and i , we deduce the vectorized function `LINSOLVETRIL_VEC` given in Algorithm 24.

Algorithm 21 Function `LINSOLVETriL` . Returns solution of equation $\mathbf{A}\mathbf{X} = \mathbf{B}$ where \mathbf{A} is a regular lower triangular matrix.

Input \mathbf{A} : in $\mathcal{M}_{d,d}(\mathbb{K})$
 \mathbf{B} : in $\mathcal{M}_{d,n}(\mathbb{K})$
Output \mathbf{X} : in $\mathcal{M}_{d,n}(\mathbb{K})$

```

1: Function  $\mathbf{X} \leftarrow \text{LINSOLVETriL}(\mathbf{A}, \mathbf{B})$ 
2: for  $l \leftarrow 1$  to  $n$  do
3:   for  $i \leftarrow 1$  to  $d$  do
4:      $S \leftarrow 0$ 
5:     for  $j \leftarrow 1$  to  $i-1$  do
6:        $S \leftarrow S + \mathbf{A}(i,j) * \mathbf{X}(j,l)$ 
7:     end for
8:      $\mathbf{X}(i,l) \leftarrow (\mathbf{B}(i,l) - S) / \mathbf{A}(i,i)$ 
9:   end for
10: end for
11: end Function

```

Algorithm 23 Function `LINSOLVETriL_CPT` , solves equation $\mathbf{A}\mathbf{X} = B$ where \mathbf{A} is a regular lower triangular 3D-array(not vectorized)

```

Function  $\mathbf{X} \leftarrow \text{LINSOLVETriL\_CPT}(\mathbf{A}, B)$ 
for  $k \leftarrow 1$  to  $N$  do
  for  $l \leftarrow 1$  to  $n$  do
    for  $i \leftarrow 1$  to  $d$  do
       $S \leftarrow 0$ 
      for  $j \leftarrow 1$  to  $i-1$  do
         $S \leftarrow S + \mathbf{A}(k,i,j) * \mathbf{X}(j,l)$ 
      end for
       $\mathbf{X}(k,i,l) \leftarrow (\text{GETCPT}(B,k,i,l) - S) / \mathbf{A}(k,i,i)$ 
    end for
  end for
end for
end Function

```

Algorithm 22 Function `LINSOLVETriL_MAT` , solves equation $\mathbf{A}\mathbf{X} = B$ where \mathbf{A} is a regular lower triangular 3D-array(not vectorized)

Input \mathbf{A} : in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$
 \mathbf{B} : in $(\mathcal{M}_{d,n}(\mathbb{K}))^N$, or in $\mathcal{M}_{d,n}(\mathbb{K})$
Output \mathbf{X} : in $(\mathcal{M}_{d,n}(\mathbb{K}))^N$

```

Function  $\mathbf{X} \leftarrow \text{LINSOLVETriL\_MAT}(\mathbf{A}, B)$ 
for  $k \leftarrow 1$  to  $N$  do
   $b \leftarrow \text{GETMAT}(B,k)$ 
   $\mathbf{X}(k, :, :) \leftarrow \text{LINSOLVETriL}(\mathbf{A}(k, :, :), b)$ 
end for
end Function

```

Algorithm 24 Function `LINSOLVETriL_VEC` , solves equation $\mathbf{A}\mathbf{X} = B$ where \mathbf{A} is a regular lower triangular 3D-array(vectorized)

```

Function  $\mathbf{X} \leftarrow \text{LINSOLVETriU\_VEC}(\mathbf{A}, B)$ 
for  $l \leftarrow 1$  to  $n$  do
  for  $i \leftarrow 1$  to  $d$  do
     $\mathbf{S} \leftarrow \text{ZEROS}(N, 1)$ 
    for  $j \leftarrow 1$  to  $i-1$  do
       $\mathbf{S} \leftarrow \mathbf{S} + \mathbf{A}(:, i, j) .* \mathbf{X}(:, j, l)$ 
    end for
     $\mathbf{X}(:, i, l) \leftarrow (\text{GETVEC}(B, i, l) - \mathbf{S}) ./ \mathbf{A}(:, i, i)$ 
  end for
end for
end Function

```

In Table 13, the computation time in second for the three functions `LINSOLVETriL_CPT` , `LINSOLVETriL_MAT` and `LINSOLVETriL_VEC` under Matlab, Octave and Python are given with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $\mathbf{B} \in (\mathcal{M}_{3,1}(\mathbb{K}))^N$. As expected the function `LINSOLVETriL_VEC` is the fastest. In Table 14, the computation time in second of the `LINSOLVETriL_VEC` function is given for N values up to 10^7 . Finally, we give in Figure 3 computational times in second of the `LINSOLVETriL_VEC` with $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$, $\mathbf{B} \in (\mathcal{M}_{d,1}(\mathbb{K}))^N$, for $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$.

3.3 Upper triangular matrices

Firstly we recall some classical formulas. Let $\mathbf{A} \in \mathcal{M}_{d,d}(\mathbb{K})$ be a regular upper triangular matrix. If $\mathbf{B} \in \mathcal{M}_{d,n}(\mathbb{K})$ then the solution $\mathbf{X} \in \mathcal{M}_{d,n}(\mathbb{K})$ of

$$\mathbf{A}\mathbf{X} = \mathbf{B} \quad (18)$$

can be computed column by column. For each column l , we successively compute $\mathbf{X}_{d,l}, \mathbf{X}_{d-1,l}, \dots, \mathbf{X}_{1,l}$ by using formula

$$\mathbf{X}_{i,l} = (\mathbf{B}_{i,l} - \sum_{j=i+1}^d \mathbf{A}_{i,j} \mathbf{X}_{j,l}) / \mathbf{A}_{i,i}, \quad \forall i \in \llbracket 1, d \rrbracket, \forall l \in \llbracket 1, n \rrbracket. \quad (19)$$

or in a more compact form we successively compute $\mathbf{X}_{d,:}, \mathbf{X}_{d-1,:}, \dots, \mathbf{X}_{1,:}$ by using formula

$$\mathbf{X}_{i,:} = (\mathbf{B}_{i,:} - \mathbf{A}_{i,i+1:d} \mathbf{X}_{i+1:d,:}) / \mathbf{A}_{i,i}, \quad \forall i \in \llbracket 1, d \rrbracket. \quad (20)$$

N	Matlab	Octave	Python	N	Matlab	Octave	Python
200 000	0.798(s)	17.990(s)	1.566(s)	200 000	2.406(s)	19.038(s)	1.649(s)
400 000	1.632(s)	36.272(s)	3.131(s)	400 000	4.790(s)	38.428(s)	3.296(s)
600 000	2.481(s)	54.323(s)	4.687(s)	600 000	7.237(s)	57.685(s)	4.954(s)
800 000	3.286(s)	72.349(s)	6.237(s)	800 000	9.657(s)	77.224(s)	6.567(s)
1 000 000	4.016(s)	90.541(s)	7.814(s)	1 000 000	11.963(s)	95.886(s)	8.273(s)

(a) Function `LINSOLVETriL_CPT`(b) Function `LINSOLVETriL_MAT`

N	Matlab	Octave	Python
200 000	0.007(s)	0.006(s)	0.008(s)
400 000	0.010(s)	0.012(s)	0.019(s)
600 000	0.022(s)	0.015(s)	0.031(s)
800 000	0.018(s)	0.023(s)	0.042(s)
1 000 000	0.022(s)	0.034(s)	0.054(s)

(c) Function `LINSOLVETriL_VEC`Table 13: Computational times in seconds of `LINSOLVETriL` functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

N	Matlab	Octave	Python
200 000	0.011(s)	0.007(s)	0.008(s)
400 000	0.010(s)	0.011(s)	0.019(s)
600 000	0.015(s)	0.021(s)	0.030(s)
800 000	0.018(s)	0.021(s)	0.041(s)
1 000 000	0.023(s)	0.027(s)	0.052(s)
5 000 000	0.255(s)	0.247(s)	0.381(s)
10 000 000	0.503(s)	0.612(s)	0.763(s)

Table 14: Computational times in seconds of the `LINSOLVETriL_VEC` function with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

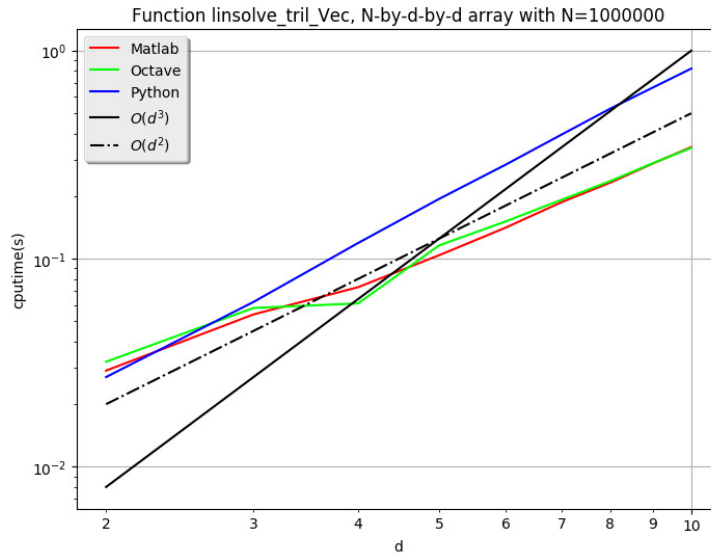


Figure 3: Computational times in seconds of the `LINSOLVETRIL_VEC` function with $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$, $\mathbf{B} \in (\mathcal{M}_{d,1}(\mathbb{K}))^N$, $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

A such operation is given by the `LINSOLVETriU` function described in Algorithm 25.

Now, one can extend these results to **regular upper triangular** 3D-array. Let $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$ be a **regular upper triangular** 3D-array, i.e. each $\mathbb{A}_k \stackrel{\text{def}}{=} \mathbf{A}(k, :, :)$ is a regular upper triangular, and so $\forall k \in \llbracket 1, N \rrbracket, \forall (i, j) \in \llbracket 1, d \rrbracket^2$

$$\begin{aligned}\mathbb{A}_k(i, j) &= 0, \text{ if } i > j \\ \mathbb{A}_k(i, i) &\neq 0.\end{aligned}$$

By using `LINSOLVETriU` and `GETMAT` functions respectively described in Algorithm 25 and Algorithm 6, we easily obtain the non-vectorized function `LINSOLVETriU_MAT` written in Algorithm 26. In Algorithm 27, an other code is presented without using function `LINSOLVETriU_MAT`. This code uses `GETCPT` function given in Algorithm 2 and by permuting the main loop in k with the two others in l and i , we deduce the vectorized function `LINSOLVETriU_VEC` given in Algorithm 28.

Algorithm 25 Function `LINSOLVETriU`. Returns solution of equation $\mathbf{A}\mathbf{X} = \mathbf{B}$ where \mathbf{A} is a regular upper triangular matrix.

Input \mathbf{A} : in $\mathcal{M}_{d,d}(\mathbb{K})$
 \mathbf{B} : in $\mathcal{M}_{d,n}(\mathbb{K})$
Output \mathbf{X} : in $\mathcal{M}_{d,n}(\mathbb{K})$

```
1: Function  $\mathbf{X} \leftarrow \text{LINSOLVETriU}(\mathbf{A}, \mathbf{B})$ 
2: for  $l \leftarrow 1$  to  $n$  do
3:   for  $i \leftarrow d$  to 1 (step -1) do
4:      $S \leftarrow 0$ 
5:     for  $j \leftarrow i + 1$  to  $d$  do
6:        $S \leftarrow S + \mathbf{A}(i, j) * \mathbf{X}(j, l)$ 
7:     end for
8:      $\mathbf{X}(i, l) \leftarrow (\mathbf{B}(i, l) - S) / \mathbf{A}(i, i)$ 
9:   end for
10: end for
11: end Function
```

Algorithm 27 Function `LINSOLVETriU_CPT`, solves equation $\mathbf{A}\mathbf{X} = \mathbf{B}$ where \mathbf{A} is a regular upper triangular 3D-array(not vectorized)

```
Function  $\mathbf{X} \leftarrow \text{LINSOLVETriU\_CPT}(\mathbf{A}, \mathbf{B})$ 
for  $k \leftarrow 1$  to  $n$  do
  for  $l \leftarrow 1$  to  $n$  do
    for  $i \leftarrow d$  to 1 (step -1) do
       $S \leftarrow 0$ 
      for  $j \leftarrow i + 1$  to  $d$  do
         $S \leftarrow S + \mathbf{A}(k, i, j) * \mathbf{X}(k, j, l)$ 
      end for
       $\mathbf{X}(k, i, l) \leftarrow (\text{GETCPT}(\mathbf{B}, k, i, l) - S) / \mathbf{A}(k, i, i)$ 
    end for
  end for
end for
end Function
```

Algorithm 26 Function `LINSOLVETriU_MAT`, solves equation $\mathbf{A}\mathbf{X} = \mathbf{B}$ where \mathbf{A} is a regular upper triangular 3D-array(not vectorized)

Input \mathbf{A} : in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$
 \mathbf{B} : in $(\mathcal{M}_{d,n}(\mathbb{K}))^N$, or in $\mathcal{M}_{d,n}(\mathbb{K})$
Output \mathbf{X} : in $(\mathcal{M}_{d,n}(\mathbb{K}))^N$

```
Function  $\mathbf{X} \leftarrow \text{LINSOLVETriU\_MAT}(\mathbf{A}, \mathbf{B})$ 
for  $k \leftarrow 1$  to  $N$  do
   $\mathbf{X}(k, :, :)$   $\leftarrow \text{LINSOLVETriU}(\mathbf{A}(k, :, :), \text{GETMAT}(\mathbf{B}, k))$ 
end for
end Function
```

Algorithm 28 Function `LINSOLVETriU_VEC`, solves equation $\mathbf{A}\mathbf{X} = \mathbf{B}$ where \mathbf{A} is a regular upper triangular 3D-array(vectorized)

```
Function  $\mathbf{X} \leftarrow \text{LINSOLVETriU\_VEC}(\mathbf{A}, \mathbf{B})$ 
for  $l \leftarrow 1$  to  $n$  do
  for  $i \leftarrow d$  to 1 (step -1) do
     $\mathbf{S} \leftarrow \text{ZEROS}(N, 1)$ 
    for  $j \leftarrow i + 1$  to  $d$  do
       $\mathbf{S} \leftarrow \mathbf{S} + \mathbf{A}(:, i, j) .* \mathbf{X}(:, j, l)$ 
    end for
     $\mathbf{X}(:, i, l) \leftarrow (\text{GETVEC}(\mathbf{B}, i, l) - \mathbf{S}) ./ \mathbf{A}(:, i, i)$ 
  end for
end for
end Function
```

In Table 15, the computation time in second for the three functions `LINSOLVETriU_CPT`, `LINSOLVETriU_MAT` and `LINSOLVETriU_VEC` under Matlab, Octave and Python are given with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $\mathbf{B} \in (\mathcal{M}_{3,1}(\mathbb{K}))^N$. As expected the function `LINSOLVETriU_VEC` is the fastest. In Table 16, the computation time in second of the `LINSOLVETriU_VEC` function is given for N values up to 10^7 . Finally, we give in Figure 4 computational times in second of the `LINSOLVETriU_VEC` with $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$, $\mathbf{B} \in (\mathcal{M}_{d,1}(\mathbb{K}))^N$, for $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$.

N	Matlab	Octave	Python	N	Matlab	Octave	Python
200 000	0.832(s)	19.640(s)	1.684(s)	200 000	2.427(s)	22.837(s)	1.789(s)
400 000	1.678(s)	39.319(s)	3.336(s)	400 000	4.853(s)	46.041(s)	3.588(s)
600 000	2.519(s)	58.020(s)	5.037(s)	600 000	7.226(s)	69.741(s)	5.361(s)
800 000	3.320(s)	79.158(s)	6.654(s)	800 000	9.649(s)	92.717(s)	7.149(s)
1 000 000	4.077(s)	101.571(s)	8.334(s)	1 000 000	12.033(s)	120.919(s)	8.941(s)

(a) Function `LINSOLVETriU_CPT`(b) Function `LINSOLVETriU_MAT`

N	Matlab	Octave	Python
200 000	0.007(s)	0.006(s)	0.008(s)
400 000	0.010(s)	0.010(s)	0.021(s)
600 000	0.021(s)	0.018(s)	0.030(s)
800 000	0.018(s)	0.029(s)	0.041(s)
1 000 000	0.023(s)	0.026(s)	0.052(s)

(c) Function `LINSOLVETriU_VEC`Table 15: Computational times in seconds of `LINSOLVETriU` functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

N	Matlab	Octave	Python
200 000	0.011(s)	0.006(s)	0.009(s)
400 000	0.011(s)	0.011(s)	0.020(s)
600 000	0.015(s)	0.018(s)	0.031(s)
800 000	0.019(s)	0.020(s)	0.042(s)
1 000 000	0.023(s)	0.026(s)	0.053(s)
5 000 000	0.255(s)	0.239(s)	0.388(s)
10 000 000	0.503(s)	0.603(s)	0.771(s)

Table 16: Computational times in seconds of the `LINSOLVETriU_VEC` function with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

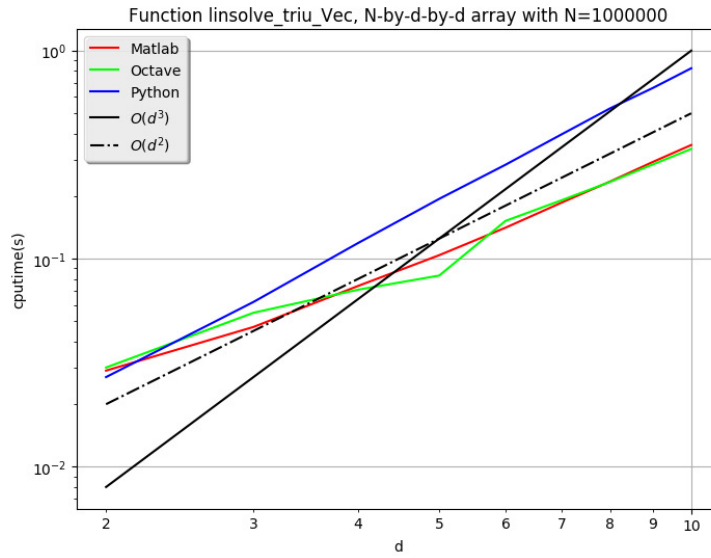


Figure 4: Computational times in seconds of the `LINSOLVETRIU_VEC` function with $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$, $\mathbf{B} \in (\mathcal{M}_{d,1}(\mathbb{K}))^N$, $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

4 Factorizations

The object of this section is to present vectorized algorithms which compute factorizations (Cholesky or LU with partial pivoting) of all matrices contained in a 3D-array.

4.1 Cholesky factorization

Firstly we recall the classical Cholesky factorization. Let \mathbb{B} be a hermitian, positive-definite matrix in $\mathcal{M}_n(\mathbb{C})$. One can apply the Cholesky factorization: there exists a unique lower triangular matrix $\mathbb{L} \in \mathcal{M}_n(\mathbb{C})$ with real and positive diagonal entries such that

$$\mathbb{B} = \mathbb{L}\mathbb{L}^*. \quad (21)$$

With this factorization, the determinant of the matrix \mathbb{B} computes easily as \mathbb{L} is a lower triangular matrix:

$$\det(\mathbb{B}) = \det(\mathbb{L})^2 = \left(\prod_{i=1}^n L_{i,i} \right)^2. \quad (22)$$

To compute the lower triangular matrix \mathbb{L} we use the following formula

$$\begin{aligned} L_{j,j} &= \sqrt{B_{j,j} - \sum_{k=1}^{j-1} L_{j,k}L_{j,k}^*} \\ L_{i,j} &= \frac{1}{L_{j,j}} \left(B_{i,j} - \sum_{k=1}^{j-1} L_{i,k}L_{j,k}^* \right) \quad \text{for } i > j. \end{aligned}$$

In Algorithm 29 the *Cholesky-Crout* algorithm is given to compute the matrix \mathbb{L} . It starts from the upper left corner of the matrix \mathbb{L} and proceeds to calculate the matrix column by column.

Algorithm 29 Function **CHOLESKY** . Computes the lower triangular matrix $\mathbb{L} \in \mathcal{M}_n(\mathbb{C})$ such that $\mathbb{B} = \mathbb{L}\mathbb{L}^*$.

Données : \mathbb{B} : a hermitian, positive-definite matrix in $\mathcal{M}_n(\mathbb{C})$.

Résultat : \mathbb{L} : the lower triangular matrix $\mathbb{L} \in \mathcal{M}_n(\mathbb{C})$
with $\mathbb{L}(i, i) > 0, \forall i \in \llbracket 1, n \rrbracket$

```

1: Function  $\mathbb{L} \leftarrow \mathbf{CHOLESKY} (\mathbb{B})$ 
2:    $\mathbb{L} \leftarrow \mathbf{ZEROS}(n, n)$ 
3:   for  $j \leftarrow 1$  to  $n$  do                                      $\triangleright$  Computes the  $j$ -th column of  $\mathbb{L}$ 
4:      $S_1 \leftarrow 0$ 
5:     for  $l \leftarrow 1$  to  $j - 1$  do
6:        $S_1 \leftarrow S_1 + |\mathbb{L}(j, l)|^2$ 
7:     end for
8:      $\mathbb{L}(j, j) \leftarrow \mathbf{SQRT}(\mathbb{B}(j, j) - S_1)$ 
9:     for  $i \leftarrow j + 1$  to  $n$  do
10:       $S_2 \leftarrow 0$ 
11:      for  $l \leftarrow 1$  to  $j - 1$  do
12:         $S_2 \leftarrow S_2 + \mathbb{L}(i, l) * \mathbf{CONJ}(\mathbb{L}(j, l))$ 
13:      end for
14:       $\mathbb{L}(i, j) \leftarrow (\mathbb{B}(i, j) - S_2) / \mathbb{L}(j, j)$ .
15:    end for
16:  end for
17: end Function

```

After these reminders, we present the heart of the matter. Let $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{C}))^N$ be a hermitian positive definite 3D-array:

$\forall k \in \llbracket 1, N \rrbracket, \mathbf{A}(k, :, :) = \mathbb{A}_k \in \mathcal{M}_n(\mathbb{C})$ is a hermitian positive definite matrix

We want to compute the lower triangular 3D-array $\mathbb{L} \in (\mathcal{M}_{d,d}(\mathbb{C}))^N$ with real and positive diagonal entries (i.e. $\forall k \in \llbracket 1, N \rrbracket, \mathbf{L}(k, :, :) = \mathbb{L}_k \in \mathcal{M}_n(\mathbb{C})$ are lower triangular matrices with real and positive diagonal entries) such that

$$\forall k \in \llbracket 1, N \rrbracket, \mathbb{A}_k = \mathbb{L}_k \mathbb{L}_k^*$$

A non-vectorized function using the function **CHOLESKY** defined in Algorithm 29 is given in Algorithm 30. To introduce the vectorized code, we firstly present in Algorithm 31 an other non-vectorized version without using the function **CHOLESKY** : this is nothing but the copy of the function in the code.

As $N \gg d$, vectorization of the Algorithm 31 consists to firstly permute the k loop over $\llbracket 1, N \rrbracket$ with the j loop over $\llbracket 1, d \rrbracket$ and then to vectorize the k loop. In this case vectorization is immediate and given in Algorithm 32. The only remaining loops are very *small* loops and not need to be vectorized.

Algorithm 30 Function `CHOLESKY_MAT` , returns cholesky factorizations of \mathbb{A}_k matrices (not vectorized)

Input \mathbf{A} : N -by- d -by- d 3D array such that
 $\mathbf{A}(k, :, :) = \mathbb{A}_k, \quad \forall k \in \llbracket 1, N \rrbracket$.

Output \mathbf{L} : a N -by- d -by- d 3D array such that
 $\forall k \in \llbracket 1, N \rrbracket, \quad \mathbb{L}_k \stackrel{\text{def}}{=} \mathbf{L}(k, :, :)$ and $\mathbb{A}_k = \mathbb{L}_k \mathbb{L}_k^*$.

```

Function  $\mathbf{L} \leftarrow \text{CHOLESKY\_MAT}(\mathbf{A})$ 
  for  $k \leftarrow 1$  to  $N$  do
     $\mathbf{L}(k, :, :) \leftarrow \text{CHOLESKY}(\mathbf{A}(k, :, :))$ 
  end for
end Function

```

Algorithm 31 Function `CHOLESKY_CPT` , returns cholesky factorizations of \mathbb{A}_k matrices (not vectorized)

```

Function  $\mathbf{L} \leftarrow \text{CHOLESKY\_CPT}(\mathbf{A})$ 
  for  $k \leftarrow 1$  to  $N$  do
    for  $j \leftarrow 1$  to  $d$  do
       $S_1 \leftarrow 0$ 
      for  $l \leftarrow 1$  to  $j - 1$  do
         $S_1 \leftarrow S_1 + |\mathbf{L}(k, j, l)|^2$ 
      end for
       $\mathbf{L}(k, j, j) \leftarrow \text{SQRT}(\mathbf{A}(k, j, j) - S_1)$ 
      for  $i \leftarrow j + 1$  to  $d$  do
         $S_2 \leftarrow 0$ 
        for  $l \leftarrow 1$  to  $j - 1$  do
           $S_2 \leftarrow S_2 + \mathbf{L}(k, i, l) * \text{CONJ}(\mathbf{L}(k, j, l))$ 
        end for
         $\mathbf{L}(k, i, j) \leftarrow (\mathbf{B}(k, i, j) - S_2) / \mathbf{L}(k, j, j)$ 
      end for
    end for
  end for
end Function

```

Algorithm 32 Function `CHOLESKY_VEC` , returns cholesky factorizations of \mathbb{A}_k matrices (vectorized)

```

1: Function  $\mathbf{L} \leftarrow \text{CHOLESKY\_VEC}(\mathbf{A})$ 
2:  $\mathbf{L} \leftarrow \text{ZEROS}(N, d, d)$ 
3: for  $j \leftarrow 1$  to  $d$  do ▷ Computes  $j$ -th column of all  $\mathbf{L}(k, :, :)$ 
4:    $\mathbf{S}_1 \leftarrow \text{ZEROS}(N)$ 
5:   for  $l \leftarrow 1$  to  $j - 1$  do
6:      $\mathbf{S}_1 \leftarrow \mathbf{S}_1 + |\mathbf{L}(:, j, l)|.^2$ 
7:   end for
8:    $\mathbf{L}(:, j, j) \leftarrow \text{SQRT}(\mathbf{A}(:, j, j) - \mathbf{S}_1)$ 
9:   for  $i \leftarrow j + 1$  to  $d$  do
10:     $\mathbf{S}_2 \leftarrow \text{ZEROS}(N)$ 
11:    for  $l \leftarrow 1$  to  $j - 1$  do
12:       $\mathbf{S}_2 \leftarrow \mathbf{S}_2 + \mathbf{L}(:, i, l) .* \text{CONJ}(\mathbf{L}(:, j, l))$ 
13:    end for
14:     $\mathbf{L}(:, i, j) \leftarrow (\mathbf{A}(:, i, j) - \mathbf{S}_2) ./ \mathbf{L}(:, j, j)$ 
15:  end for
16: end for
17: end Function

```

In Table 17, computational time in second for the three functions `CHOLESKY_CPT` , `CHOLESKY_MAT` and `CHOLESKY_VEC` under Matlab, Octave and Python are given with input data in $(\mathcal{M}_{3,3}(\mathbb{K}))^N$ and for N up to 10^5 . As expected the function `CHOLESKY_VEC` is the fastest. In Table 18, computational time in second of the `CHOLESKY_VEC` function is given for N values up to 10^7 . Furthermore the `numpy.linalg.cholesky` Python function natively support Cholesky factorization on 3D-arrays and we added its computational times in Table 18 under the reference `Python[Mat]`. As we can see Matlab performs better but this is partially due to its multithreading capacities (see Table 19). Finally, we give in Figure 5 the computation time in second of the `CHOLESKY_VEC` with input data in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$ for $N = 10^6$ and $d \in [2, 10]$.

4.2 LU factorization with partial pivoting

At first we briefly recall results on LU factorization with partial pivoting. Let $\mathbf{A} \in \mathcal{M}_d(\mathbb{C})$ (not necessarily regular). The LU factorization with partial pivoting, described in [1], [2], is a well know method which allows to compute a permutation matrix $\mathbb{P} \in \mathcal{M}_d(\mathbb{R})$, a lower triangular matrix $\mathbf{L} \in \mathcal{M}_d(\mathbb{C})$ with unit diagonal and an upper triangular matrix $\mathbf{U} \in \mathcal{M}_d(\mathbb{C})$ such that

$$\mathbb{P}\mathbf{A} = \mathbf{L}\mathbf{U}. \quad (23)$$

In Algorithm 33, a classical computation of the three matrices \mathbb{P} , \mathbf{L} and \mathbf{U} is proposed. Thereafter, the less memory consuming Algorithm 34 is given where \mathbf{U} is stored in the upper triangle of \mathbf{A} and \mathbf{L} in the strictly lower triangle of \mathbf{A} .

The object of this section is to describe a vectorized version of these two algorithms apply to 3D-arrays. More precisely, let $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$, we want to compute the three 3D-arrays \mathbb{P} , \mathbf{L} and \mathbf{U} in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$ such that, for all

N	Matlab	Octave	Python	N	Matlab	Octave	Python
20 000	0.212(s)	2.985(s)	0.244(s)	20 000	0.287(s)	3.238(s)	0.250(s)
40 000	0.425(s)	6.010(s)	0.491(s)	40 000	0.564(s)	6.523(s)	0.508(s)
60 000	0.614(s)	9.005(s)	0.734(s)	60 000	0.837(s)	9.777(s)	0.745(s)
80 000	0.835(s)	12.016(s)	0.996(s)	80 000	1.104(s)	13.035(s)	0.994(s)
100 000	1.004(s)	15.004(s)	1.233(s)	100 000	1.389(s)	16.337(s)	1.247(s)

(a) Function `CHOLESKY_CPT`(b) Function `CHOLESKY_MAT`

N	Matlab	Octave	Python
20 000	0.002(s)	0.001(s)	0.001(s)
40 000	0.002(s)	0.001(s)	0.003(s)
60 000	0.002(s)	0.002(s)	0.004(s)
80 000	0.003(s)	0.003(s)	0.005(s)
100 000	0.004(s)	0.004(s)	0.007(s)

(c) Function `CHOLESKY_VEC`Table 17: Computational times in seconds of `CHOLESKY` functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

N	Matlab	Matlab(*)	Octave	Python	Python(Nat)
200 000	0.009(s)	0.010(s)	0.008(s)	0.019(s)	0.018(s)
400 000	0.016(s)	0.019(s)	0.016(s)	0.043(s)	0.036(s)
600 000	0.025(s)	0.031(s)	0.035(s)	0.074(s)	0.059(s)
800 000	0.033(s)	0.042(s)	0.047(s)	0.100(s)	0.089(s)
1 000 000	0.042(s)	0.054(s)	0.061(s)	0.129(s)	0.110(s)
5 000 000	0.329(s)	0.445(s)	0.523(s)	0.809(s)	0.542(s)
10 000 000	0.655(s)	0.904(s)	1.036(s)	1.615(s)	1.079(s)

Table 18: Computational times in seconds of the `CHOLESKY_VEC` function with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

N	1 threads	2 threads	4 threads	6 threads	8 threads	14 threads	20 threads	28 threads
200 000	0.010(s)	0.008(s)	0.008(s)	0.008(s)	0.007(s)	0.007(s)	0.007(s)	0.008(s)
400 000	0.017(s)	0.016(s)	0.015(s)	0.015(s)	0.014(s)	0.014(s)	0.014(s)	0.014(s)
600 000	0.031(s)	0.029(s)	0.027(s)	0.026(s)	0.026(s)	0.025(s)	0.025(s)	0.025(s)
800 000	0.047(s)	0.043(s)	0.040(s)	0.038(s)	0.037(s)	0.036(s)	0.036(s)	0.036(s)
1 000 000	0.054(s)	0.050(s)	0.046(s)	0.044(s)	0.043(s)	0.041(s)	0.041(s)	0.041(s)
5 000 000	0.445(s)	0.398(s)	0.368(s)	0.351(s)	0.344(s)	0.331(s)	0.327(s)	0.323(s)
10 000 000	0.895(s)	0.780(s)	0.709(s)	0.692(s)	0.677(s)	0.658(s)	0.651(s)	0.645(s)

Table 19: Function `CHOLESKY_VEC` with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ under Matlab 2018a: effect of multithreaded on cputimes

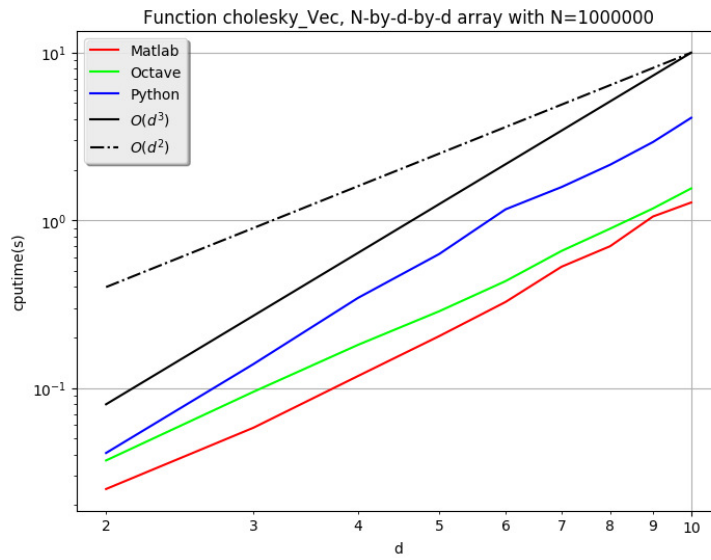


Figure 5: Computational times in seconds of `CHOLESKY_VEC` function with $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$ with $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

$k \in \llbracket 1, N \rrbracket$, a LU factorization with partial pivoting of the matrix $A_k = \mathbf{A}(k, :, :)$ is given by

$$\mathbb{P}_k A_k = \mathbb{L}_k \mathbb{U}_k$$

where $\mathbb{P}_k = \mathbf{P}(k, :, :)$, $\mathbb{L}_k = \mathbf{L}(k, :, :)$ and $\mathbb{U}_k = \mathbf{U}(k, :, :)$.

Algorithm 33 Function **PALU** computes the LU factorization with partial pivoting of a matrix A such that $\mathbb{P}A = \mathbb{L}\mathbb{U}$.

Données : A : matrix in $\mathcal{M}_d(\mathbb{K})$.
Résultat : \mathbb{P} : permutation matrix in $\mathcal{M}_d(\mathbb{K})$,
 \mathbb{L} : lower triangular matrix in $\mathcal{M}_d(\mathbb{K})$
avec $\mathbb{L}(i, i) = 1, \forall i \in \llbracket 1, d \rrbracket$,
 \mathbb{U} : upper triangular matrix in $\mathcal{M}_d(\mathbb{K})$.

```

1: Function [ $\mathbb{P}, \mathbb{L}, \mathbb{U}$ ]  $\leftarrow$  PALU ( $A$ )
2:  $\mathbb{P} \leftarrow$  EYE( $d$ ),  $\mathbb{L} \leftarrow$  EYE( $d$ ),  $\mathbb{U} \leftarrow A$ 
3: for  $i \leftarrow 1$  to  $d$  do
4:    $\mu \leftarrow$  ARGMAX( $|\mathbb{U}(i : d, i)|$ ) + ( $i - 1$ )
5:   if  $|\mathbb{U}(\mu, i)| > \epsilon$  then
6:     if  $\mu \neq i$  then                                      $\triangleright$  Permutes rows  $i$  and  $\mu$ 
7:        $\mathbb{U}(i, i : d) \leftrightarrow \mathbb{U}(\mu, i : d)$                     $\triangleright$  Only columns  $i$  to  $d$ 
8:        $\mathbb{L}(i, 1 : i - 1) \leftrightarrow \mathbb{L}(\mu, 1 : i - 1)$           $\triangleright$  Only columns 1 to  $i - 1$ 
9:        $\mathbb{P}(i, :) \leftrightarrow \mathbb{P}(\mu, :)$                             $\triangleright$  All columns
10:    end if
11:    for  $j \leftarrow i + 1$  to  $d$  do
12:       $\mathbb{L}(j, i) \leftarrow \mathbb{U}(j, i) / \mathbb{U}(i, i)$ 
13:       $\mathbb{L}(j, i : d) \leftarrow \mathbb{U}(j, i : d) - \mathbb{L}(j, i) * \mathbb{U}(i, i : nd)$ 
14:    end for
15:  end if
16: end for
17: end Function

```

Algorithm 34 Function **PLUINPLACE** inplace computation of the LU factorization with partial pivoting of a matrix \mathbb{A} such that $\mathbb{P}\mathbb{A} = \mathbb{L}\mathbb{U}$.

Données : \mathbb{A} : matrix in $\mathcal{M}_n(\mathbb{K})$.
Résultat : \mathbf{p} : rows permutation index
 \mathbb{A} : the modified matrix such that ...

```

1: Function [ $\mathbf{p}, \mathbb{A}$ ]  $\leftarrow$  PLUINPLACE (  $\mathbb{A}$  )
2:    $\mathbf{p} \leftarrow 1 : n$ 
3:   for  $i \leftarrow 1$  to  $n - 1$  do
4:      $\mu \leftarrow \text{ARGMAX}(\mathbb{A}(i : n, i)) + (i - 1)$ 
5:     if  $|\mathbb{A}(\mu, i)| > \epsilon$  then
6:       if  $\mu \neq i$  then                                      $\triangleright$  Permutes rows  $i$  and  $\mu$ 
7:          $\mathbb{A}(i, :) \leftrightarrow \mathbb{A}(\mu, :)$ 
8:          $\mathbf{p}(i) \leftrightarrow \mathbf{p}(\mu)$ 
9:       end if
10:       $I \leftarrow i + 1 : n$ 
11:       $\mathbb{A}(I, i) \leftarrow \mathbb{A}(I, i) / \mathbb{A}(i, i)$ 
12:       $\mathbb{A}(I, I) \leftarrow \mathbb{A}(I, I) - \mathbb{A}(I, i) * \mathbb{A}(i, I)$ 
13:    end if
14:  end for
15: end Function

```

Unlike the cholesky decomposition, there is currently no Numpy o Scipy Python function calculating a LU factorization for an 3D-array.

4.2.1 Full computation

Firstly, we give Algorithm 35 a trivial but not vectorized version of the Algorithm 33 apply to the 3D-array $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$. This code use on each matrix $\mathbf{A}(k, :, :)$ the function **PALU** described in Algorithm 33. Before vectorizing, one have to write the complet code without using this function. This is done in Algorithm 36.

Algorithm 35 Function **PALU_MAT** computes all LU factorizations with partial pivoting of a 3D-array \mathbf{A} such that $\mathbb{P}_k \mathbf{A}_k = \mathbb{L}_k \mathbf{U}_k$

Input \mathbf{A} : in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$.

Output \mathbf{P} : permutation matrices in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$.
 \mathbf{L} : lower triangular matrices in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$
avec $\mathbb{L}_k(i, i) = 1, \forall i \in \llbracket 1, d \rrbracket$,
 \mathbf{U} : upper triangular matrices in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$.

```

1: Function [ $\mathbf{P}, \mathbf{L}, \mathbf{U}$ ]  $\leftarrow$  PALU_MAT (  $\mathbf{A}$  )
2:   for  $k \leftarrow 1$  to  $N$  do
3:     [ $\mathbf{P}(k, :, :), \mathbf{L}(k, :, :), \mathbf{U}(k, :, :)$ ]  $\leftarrow$  PALU( $\mathbf{A}(k, :, :)$ )
4:   end for
5: end Function

```

Algorithm 36 Function `PALU_CPT` computes all LU factorizations with partial pivoting of a 3D-array \mathbf{A} such that $\mathbb{P}_k \mathbf{A}_k = \mathbb{L}_k \mathbb{U}_k$

```

1: Function [ $\mathbf{P}, \mathbf{L}, \mathbf{U}$ ]  $\leftarrow$  PALU_CPT ( $\mathbf{A}$ )
2:   for  $k \leftarrow 1$  to  $N$  do
3:     for  $i \leftarrow 1$  to  $d$  do
4:        $\mu \leftarrow \text{ARGMAX}(|\mathbf{U}(k, i : d, i)|) + (i - 1)$ 
5:       if  $|\mathbf{U}(k, \mu, i)| > \epsilon$  then
6:         if  $\mu \neq i$  then  $\triangleright$  Permutes rows  $i$  and  $\mu$ 
7:            $\mathbf{U}(k, i, i : d) \leftrightarrow \mathbf{U}(k, \mu, i : d)$   $\triangleright$  Only columns  $i$  to  $d$ 
8:            $\mathbf{L}(k, i, 1 : i - 1) \leftrightarrow \mathbf{L}(k, \mu, 1 : i - 1)$   $\triangleright$  Only columns 1 to  $i - 1$ 
9:            $\mathbf{P}(k, i, :) \leftrightarrow \mathbf{P}(k, \mu, :)$   $\triangleright$  All columns
10:        end if
11:        for  $j \leftarrow i + 1$  to  $d$  do  $\triangleright$  Elimination
12:           $\mathbf{L}(k, j, i) \leftarrow \mathbf{U}(k, j, i) / \mathbf{U}(k, i, i)$ 
13:           $\mathbf{L}(k, j, i : d) \leftarrow \mathbf{U}(k, j, i : d) - \mathbf{L}(k, j, i) * \mathbf{U}(k, i, i : nd)$ 
14:        end for
15:      end if
16:    end for
17:  end for
18: end Function

```

As $N \gg n$, the vectorization of the Algorithm 36 consists in *removing* the k loop over the N matrices. Thereafter we permute the i loop with the k loop. Let $i \in \llbracket 1, d \rrbracket$, finding all pivoting index for each matrix on column i and rows $i : d$ on a not vectorized way is:

$$\boldsymbol{\mu}(k) \leftarrow \text{ARGMAX}(|\mathbf{U}(k, i : d, i)|) + (i - 1), \quad \forall k \in \llbracket 1, N \rrbracket$$

Vectorization is obtained via the `ARGMAX` function by searching arg max values on 2D-array $|\mathbf{U}(:, i : d, i)|$ along 2-nd dimension (axis):

$$\boldsymbol{\mu} \leftarrow \text{ARGMAX}(|\mathbf{U}(:, i : d, i)|, 2) + (i - 1).$$

Permutation at step i for \mathbf{P} is

$$\mathbf{P}(k, i, :) \leftrightarrow \mathbf{P}(k, \boldsymbol{\mu}(k), :), \quad \forall k \in \llbracket 1, N \rrbracket$$

This can be written as

```

for  $j \leftarrow 1$  to  $d$  do
   $\mathbf{P}(k, i, j) \leftrightarrow \mathbf{P}(k, \boldsymbol{\mu}(k), j), \quad \forall k \in \llbracket 1, N \rrbracket$ 
end for

```

So there are $N \times d$ permutations to do and thus we must use linear index access to \mathbf{P} elements for vectorizing a such operation.

```

 $J \leftarrow \text{ONES}(1, N)$ 
for  $j \leftarrow 1$  to  $d$  do
   $I_1 \leftarrow \text{SUB2IND}(\llbracket N, d, d \rrbracket, 1 : N, \boldsymbol{\mu}, j * J)$ 
   $I_2 \leftarrow \text{SUB2IND}(\llbracket N, d, d \rrbracket, 1 : N, i * J, j * J)$ 
   $\mathbf{P}(I_1) \leftrightarrow \mathbf{P}(I_2)$ 
end for

```

Permutation rows for \mathbf{L} and \mathbf{U} are obtained in a similar way.

Finally the elimination loop (Algorithm 36, lines 11 to 14) is only done for all indices $k \in \llbracket 1, N \rrbracket$ such that $|U(k, i, i)| > \epsilon$ and we obtain the following vectorization:

```

K ← |U(:, i, i)| > ε
for j ← i + 1 to d do
  L(K, j, i) ← U(K, j, i) ./ U(K, i, i)
  L(K, j, i : d) ← U(K, j, i : d) - L(K, j, i) .* U(K, i, i : d)
end for

```

The complete vectorized code is given by `PALU_VEC` function in Algorithm 37.

Algorithm 37 Function `PALU_VEC` computes all LU factorizations with partial pivoting of a 3D-array \mathbf{A} such that $\mathbb{P}_k \mathbf{A}_k = \mathbf{L}_k \mathbf{U}_k$

```

1: Function [P, L, U] ← PALU_VEC ( A )
2:   K ← 1 : N
3:   J ← ONES(1, N)
4:   for i ← 1 to d do
5:     μ ← ARGMAX(|U(:, i : d, i)|, 2) + (i - 1)
6:     for j ← 1 to i - 1 do
7:       I1 ← SUB2IND([N, d, d], K, μ, j * J)
8:       I2 ← SUB2IND([N, d, d], K, i * J, j * J)
9:       L(I1) ↔ L(I2)
10:      P(I1) ↔ P(I2)
11:    end for
12:    for j ← i + 1 to d do
13:      I1 ← SUB2IND([N, d, d], K, μ, j * J)
14:      I2 ← SUB2IND([N, d, d], K, i * J, j * J)
15:      U(I1) ↔ U(I2)
16:      P(I1) ↔ P(I2)
17:    end for
18:    Kidx ← |U(:, i, i)| > ε
19:    for j ← i + 1 to d do
20:      L(Kidx, j, i) ← U(Kidx, j, i) ./ U(Kidx, i, i)
21:      L(Kidx, j, i : d) ← U(Kidx, j, i : d) - L(Kidx, j, i) .* U(Kidx, i, i : d)
22:    end for
23:  end for
24: end Function

```

In Table 20, the computation time in second for the three functions `PALU_CPT`, `PALU_MAT` and `PALU_VEC` under Matlab, Octave and Python are given with input data in $(\mathcal{M}_{3,3}(\mathbb{K}))^N$. As expected the function `PALU_VEC` is the fastest. In Table 21, the computation time in second for `PALU_VEC` is given for N values up to 10^7 . Finally, we give in Figure 6 the computation time in second of the `PALU_VEC` with input data in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$ for $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$.

N	Matlab	Octave	Python	N	Matlab	Octave	Python
200 000	3.471(s)	42.204(s)	6.138(s)	200 000	4.447(s)	56.453(s)	7.184(s)
400 000	7.001(s)	84.588(s)	12.487(s)	400 000	8.927(s)	112.811(s)	14.330(s)
600 000	10.471(s)	127.051(s)	18.614(s)	600 000	13.413(s)	169.860(s)	21.311(s)
800 000	13.968(s)	169.286(s)	25.095(s)	800 000	17.932(s)	223.056(s)	28.140(s)
1 000 000	17.370(s)	209.688(s)	31.455(s)	1 000 000	22.506(s)	278.533(s)	35.551(s)

(a) Function `PALU_CPT`(b) Function `PALU_MAT`

N	Matlab	Octave	Python
200 000	0.082(s)	0.075(s)	0.127(s)
400 000	0.168(s)	0.159(s)	0.271(s)
600 000	0.253(s)	0.265(s)	0.446(s)
800 000	0.342(s)	0.384(s)	0.627(s)
1 000 000	0.431(s)	0.467(s)	0.811(s)

(c) Function `PALU_VEC`Table 20: Computational times in seconds of `PALU` functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

N	Matlab	Matlab(*)	Octave	Python
200 000	0.090(s)	0.091(s)	0.076(s)	0.129(s)
400 000	0.209(s)	0.207(s)	0.184(s)	0.274(s)
600 000	0.251(s)	0.308(s)	0.268(s)	0.416(s)
800 000	0.339(s)	0.426(s)	0.388(s)	0.562(s)
1 000 000	0.447(s)	0.535(s)	0.480(s)	0.770(s)
5 000 000	3.798(s)	4.428(s)	3.977(s)	5.650(s)
10 000 000	7.374(s)	8.934(s)	8.138(s)	11.292(s)

Table 21: Computational times in seconds of `PALU_VEC` functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

N	1 threads	2 threads	4 threads	6 threads	8 threads	14 threads	20 threads	28 threads
200 000	0.093(s)	0.095(s)	0.084(s)	0.082(s)	0.079(s)	0.077(s)	0.077(s)	0.078(s)
400 000	0.206(s)	0.197(s)	0.179(s)	0.170(s)	0.166(s)	0.166(s)	0.164(s)	0.164(s)
600 000	0.308(s)	0.304(s)	0.273(s)	0.261(s)	0.257(s)	0.250(s)	0.255(s)	0.254(s)
800 000	0.442(s)	0.433(s)	0.409(s)	0.395(s)	0.384(s)	0.386(s)	0.386(s)	0.392(s)
1 000 000	0.532(s)	0.516(s)	0.467(s)	0.447(s)	0.438(s)	0.427(s)	0.439(s)	0.442(s)
5 000 000	4.424(s)	4.052(s)	3.833(s)	3.769(s)	3.727(s)	3.696(s)	3.752(s)	3.742(s)
10 000 000	8.951(s)	8.086(s)	7.593(s)	7.466(s)	7.385(s)	7.379(s)	7.496(s)	7.406(s)

Table 22: Function `PALU_VEC` with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ under Matlab 2018a: effect of multithreaded on cputimes

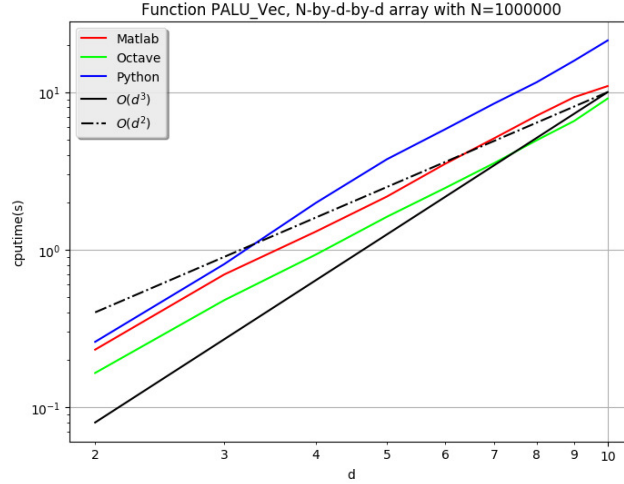


Figure 6: Computational times in seconds of `PALU_VEC` function with $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$ with $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

4.2.2 Inplace computation

From Algorithm 34, we immediately obtain the not vectorized Algorithm 38 for a 3D-array $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$. Before vectorizing, one have to write the complet code without using this function. This is done in Algorithm 39.

Algorithm 38 Function `pLUINPLACE_MAT` computes all LU factorizations with partial pivoting of a 3D-array \mathbf{A} such that $\mathbb{P}_k \mathbf{A}_k = \mathbb{L}_k \mathbb{U}_k$

Input \mathbf{A} : in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$.

Output \mathbb{p} : permutation index array in $(\mathcal{M}_d(\mathbb{K}))^N$.
 \mathbf{A} : the modified 3D-array

```

1: Function [ $\mathbb{p}, \mathbf{A}$ ]  $\leftarrow$  pLUINPLACE_MAT (  $\mathbf{A}$  )
2:   for  $k \leftarrow 1$  to  $N$  do
3:     [ $\mathbb{p}(k, :), \mathbf{A}(k, :, :)$ ]  $\leftarrow$  pLUINPLACE( $\mathbf{A}(k, :, :)$ )
4:   end for
5: end Function

```

Algorithm 39 Function `PLUINPLACE_CPT` computes all LU factorizations with partial pivoting of a 3D-array \mathbf{A} such that $\mathbb{P}_k \mathbf{A}_k = \mathbb{L}_k \mathbf{U}_k$

```

1: Function [ $\mathbb{p}, \mathbf{A}$ ]  $\leftarrow$  PLUINPLACE_CPT (  $\mathbf{A}$  )
2:    $\mathbb{p} \leftarrow$  REPTILE(1 :  $n, N, 1$ )
3:   for  $k \leftarrow 1$  to  $N$  do
4:     for  $i \leftarrow 1$  to  $n - 1$  do
5:        $\mu \leftarrow$  ARGMAX( $\mathbf{A}(k, i : n, i)$ ) + ( $i - 1$ )
6:       if  $|\mathbf{A}(k, \mu, i)| > \epsilon$  then
7:         if  $\mu \neq i$  then  $\triangleright$  Permutes rows  $i$  and  $\mu$ 
8:            $\mathbf{A}(k, i, :) \leftrightarrow \mathbf{A}(k, \mu, :)$ 
9:            $\mathbb{p}(k, i) \leftrightarrow \mathbb{p}(k, \mu)$ 
10:        end if
11:        $I \leftarrow i + 1 : n$ 
12:        $\mathbf{A}(k, I, i) \leftarrow \mathbf{A}(k, I, i) / \mathbf{A}(k, i, i)$ 
13:        $\mathbf{A}(k, I, I) \leftarrow \mathbf{A}(k, I, I) - \mathbf{A}(k, I, i) * \mathbf{A}(k, i, I)$ 
14:     end if
15:   end for
16: end for
17: end Function

```

The complete vectorized code is given by `PLUINPLACE_VEC` function in Algorithm 40 and is close to the `PALU_VEC` function given in Algorithm 37.

Algorithm 40 Function `PLUINPLACE_VEC` computes all LU factorizations with partial pivoting of a 3D-array \mathbf{A} such that $\mathbb{P}_k \mathbf{A}_k = \mathbb{L}_k \mathbf{U}_k$

```

1: Function [ $\mathbb{p}, \mathbf{A}$ ]  $\leftarrow$  PLUINPLACE_VEC (  $\mathbf{A}$  )
2:    $\mathbb{p} \leftarrow$  REPTILE(1 :  $d, N, 1$ )
3:    $\mathbf{K} \leftarrow 1 : N$ 
4:    $\mathbf{J} \leftarrow$  ONES(1,  $N$ )
5:   for  $i \leftarrow 1$  to  $n - 1$  do
6:      $\mu \leftarrow$  ARGMAX( $|\mathbf{U}(:, i : d, i)|, 2$ ) + ( $i - 1$ )
7:     for  $j \leftarrow 1$  to  $d$  do
8:        $I_1 \leftarrow$  SUB2IND( $[N, d, d], \mathbf{K}, \mu, j * \mathbf{J}$ )
9:        $I_2 \leftarrow$  SUB2IND( $[N, d, d], \mathbf{K}, i * \mathbf{J}, j * \mathbf{J}$ )
10:       $\mathbf{A}(I_1) \leftrightarrow \mathbf{A}(I_2)$ 
11:    end for
12:     $I_1 \leftarrow$  SUB2IND( $[N, d], \mathbf{K}, \mu$ )
13:     $I_2 \leftarrow$  SUB2IND( $[N, d], \mathbf{K}, i * \mathbf{J}$ )
14:     $\mathbb{p}(I_1) \leftrightarrow \mathbb{p}(I_2)$ 
15:     $\mathbf{Kidx} \leftarrow |\mathbf{U}(:, i, i)| > \epsilon$ 
16:    for  $j \leftarrow i + 1$  to  $d$  do
17:       $\mathbf{L}(\mathbf{Kidx}, j, i) \leftarrow \mathbf{U}(\mathbf{Kidx}, j, i) ./ \mathbf{U}(\mathbf{Kidx}, i, i)$ 
18:       $\mathbf{L}(\mathbf{Kidx}, j, i : d) \leftarrow \mathbf{U}(\mathbf{Kidx}, j, i : d) - \mathbf{L}(\mathbf{Kidx}, j, i) .* \mathbf{U}(\mathbf{Kidx}, i, i : d)$ 
19:    end for
20:  end for
21: end Function

```

In Table 23, the computation time in second for the three functions `pLUINPLACE_CPT`, `pLUINPLACE_MAT` and `pLUINPLACE_VEC` under Matlab, Octave and Python are given with input data in $(\mathcal{M}_{3,3}(\mathbb{K}))^N$. As expected the function `pLUINPLACE_VEC` is the fastest. In Table 24, the computation time in second for `pLUINPLACE_VEC` is given for N values up to 10^7 . In Table 25 effects of multithreading on cputimes is given. Finally, we give in Figure 7 the computation time in second of the `pLUINPLACE_VEC` with input data in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$ for $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$.

N	Matlab	Octave	Python	N	Matlab	Octave	Python
200 000	3.701(s)	30.816(s)	4.099(s)	200 000	4.168(s)	37.104(s)	4.060(s)
400 000	7.508(s)	63.312(s)	8.115(s)	400 000	8.376(s)	74.818(s)	8.086(s)
600 000	11.289(s)	94.744(s)	12.259(s)	600 000	12.572(s)	111.888(s)	12.202(s)
800 000	14.960(s)	126.546(s)	16.281(s)	800 000	16.668(s)	146.495(s)	16.207(s)
1 000 000	18.616(s)	155.061(s)	20.352(s)	1 000 000	20.805(s)	182.749(s)	20.289(s)

(a) Function `pLUINPLACE_CPT`

(b) Function `pLUINPLACE_MAT`

N	Matlab	Octave	Python
200 000	0.069(s)	0.080(s)	0.070(s)
400 000	0.118(s)	0.175(s)	0.159(s)
600 000	0.203(s)	0.320(s)	0.252(s)
800 000	0.303(s)	0.405(s)	0.362(s)
1 000 000	0.365(s)	0.479(s)	0.470(s)

(c) Function `pLUINPLACE_VEC`

Table 23: Computational times in seconds of `pLUINPLACE` functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

N	Matlab	Matlab(*)	Octave	Python
200 000	0.071(s)	0.079(s)	0.083(s)	0.070(s)
400 000	0.140(s)	0.169(s)	0.169(s)	0.167(s)
600 000	0.198(s)	0.265(s)	0.299(s)	0.284(s)
800 000	0.287(s)	0.342(s)	0.420(s)	0.372(s)
1 000 000	0.356(s)	0.500(s)	0.500(s)	0.492(s)
5 000 000	2.897(s)	3.784(s)	4.335(s)	3.234(s)
10 000 000	5.628(s)	7.559(s)	8.489(s)	6.406(s)

Table 24: Computational times in seconds of `pLUINPLACE_VEC` functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

N	1 threads	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	14 threads	18 threads	24 threads
200 000	0.080(s)	0.080(s)	0.067(s)	0.067(s)	0.065(s)	0.063(s)	0.063(s)	0.063(s)	0.063(s)	0.063(s)
400 000	0.171(s)	0.158(s)	0.133(s)	0.127(s)	0.121(s)	0.121(s)	0.120(s)	0.123(s)	0.119(s)	0.120(s)
600 000	0.270(s)	0.266(s)	0.240(s)	0.226(s)	0.219(s)	0.216(s)	0.215(s)	0.214(s)	0.221(s)	0.218(s)
800 000	0.416(s)	0.409(s)	0.314(s)	0.304(s)	0.294(s)	0.293(s)	0.287(s)	0.291(s)	0.295(s)	0.292(s)
1 000 000	0.472(s)	0.455(s)	0.408(s)	0.387(s)	0.375(s)	0.370(s)	0.365(s)	0.367(s)	0.375(s)	0.375(s)
5 000 000	3.759(s)	3.309(s)	3.017(s)	2.938(s)	2.906(s)	2.865(s)	2.844(s)	2.848(s)	2.899(s)	2.895(s)
10 000 000	7.465(s)	6.513(s)	5.945(s)	5.793(s)	5.730(s)	5.681(s)	5.661(s)	5.643(s)	5.755(s)	5.665(s)

Table 25: Function `pLUINPLACE_VEC` with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ under Matlab 2018a: effect of multithreaded on cputimes

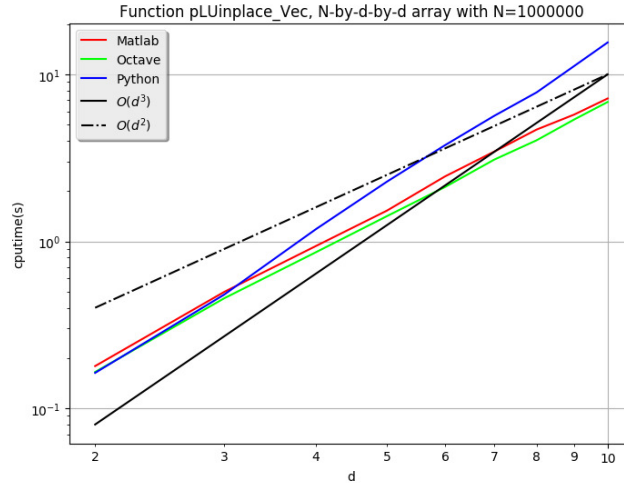


Figure 7: Computational times in seconds of `pLUINPLACE_VEC` function with $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$ with $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

5 Linear solvers

To solve linear system one can use the LU factorization with partial pivoting for regular matrices or the cholesky factorization for symmetric positive definite matrices.

Let $B \in (\mathcal{M}_{d,n}(\mathbb{K}))^N$ or $B \in \mathcal{M}_{d,n}(\mathbb{K})$, we want to solve the equation

$$\mathbf{A}\mathbf{X} = B$$

as described in section 1.2.3

5.1 Using Cholesky factorization

Let $\mathbf{A} \in \mathcal{M}_{d,d}(\mathbb{K})$ be symmetric positive definite matrix and $B \in \mathcal{M}_{d,n}(\mathbb{K})$. As seen in 4.1, there exists an unique lower triangular matrix \mathbb{L} with strictly positive diagonal elements such that $\mathbf{A} = \mathbb{L}\mathbb{L}^*$. So, to solve the equation

$$\mathbf{A}\mathbf{X} = B$$

one just have to solve the two triangular systems

$$\mathbb{L}\mathbf{Y} = B \text{ then } \mathbb{L}^*\mathbf{X} = \mathbf{Y}.$$

For a symmetric positive definite 3D-array we immediatly have the `LIN-SOLVECHOLESKY_VEC` vectorized function given in Algorithm 41. This function

uses the vectorized functions `CHOLESKY_VEC`, `LINSOLVETRIL_VEC`, `LINSOLVETRIU_VEC` and `CTRANSPOSE_VEC` respectively given in Algorithms 32, 28, 28 and ???. In Table 26, computational times in second with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $B \in (\mathcal{M}_{3,1}(\mathbb{K}))^N$ under Matlab, Octave and Python are given for N values up to 10^7 . In Figure 8 the computation time in second of the `LINSOLVECHOLESKY_VEC` with $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$ and $B \in (\mathcal{M}_{d,1}(\mathbb{K}))^N$ for $N = 10^6$ and $d \in [2, 10]$ is represented.

Algorithm 41 Function `LINSOLVECHOLESKY_VEC`, solves equation $\mathbf{A}\mathbf{X} = B$ where \mathbf{A} is a symmetric positive definite 3D-array (vectorized)

```

Function  $\mathbf{X} \leftarrow \text{LINSOLVECHOLESKY\_VEC}(\mathbf{A}, B)$ 
   $\mathbf{L} \leftarrow \text{CHOLESKY\_VEC}(A)$ 
   $\mathbf{Y} \leftarrow \text{LINSOLVETRIL\_VEC}(\mathbf{L}, B)$ 
   $\mathbf{X} \leftarrow \text{LINSOLVETRIU\_VEC}(\text{CTRANSPOSE\_VEC}(\mathbf{L}), \mathbf{Y})$ 
end Function

```

N	Matlab	Octave	Python
200 000	0.024(s)	0.023(s)	0.037(s)
400 000	0.047(s)	0.041(s)	0.080(s)
600 000	0.071(s)	0.084(s)	0.132(s)
800 000	0.092(s)	0.111(s)	0.176(s)
1 000 000	0.117(s)	0.142(s)	0.221(s)
5 000 000	0.786(s)	1.300(s)	1.441(s)
10 000 000	1.789(s)	2.589(s)	2.868(s)

Table 26: Computational times in seconds of the `LINSOLVECHOLESKY_VEC` function with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $B \in (\mathcal{M}_{3,1}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

5.2 Using LU factorization with partial pivoting

Let $A \in \mathcal{M}_{d,d}(\mathbb{K})$ be a regular matrix and $B \in \mathcal{M}_{d,n}(\mathbb{K})$. As seen in 4.2, there exists a permutation matrix P , a lower triangular matrix L with unit diagonal and an upper triangular matrix U such that $PA = LU$. So, to solve the equation

$$A\mathbf{X} = B$$

one just have to solve the two triangular systems

$$LY = B \quad \text{then} \quad UX = Y.$$

For a regular 3D-array we immediatly deduce the `LINSOLVEPALU_VEC` vectorized function given in vectorized Algorithm 42. This function uses the vectorized functions `PALU_VEC`, `LINSOLVETRIL_VEC` and `LINSOLVETRIU_VEC` respectively given in Algorithms 37, 24 and 28. In Table 27, computational times in second with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $B \in (\mathcal{M}_{3,1}(\mathbb{K}))^N$ under Matlab, Octave and Python are given for N values up to 10^7 . Furthermore with Python, the broadcasting rules can be applied by using the `numpy.linalg.solve` function that we

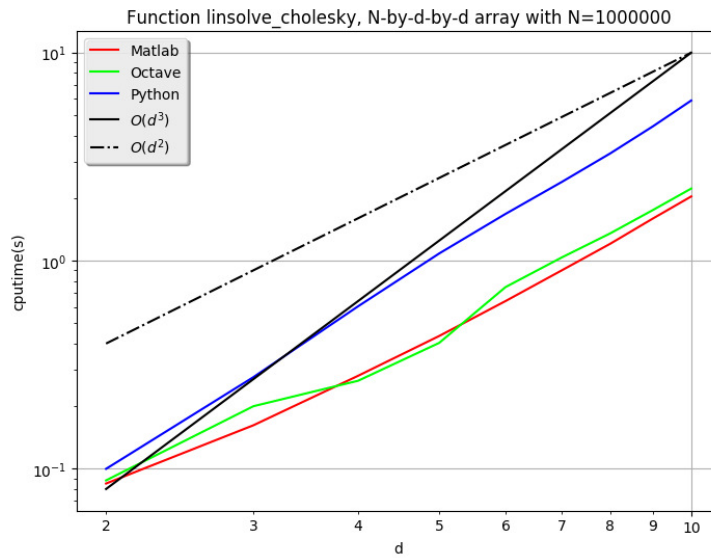


Figure 8: Computational times in seconds of `LINSOLVECHOLESKY_VEC` function with $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$ with $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

denote by «Python[Nat]» in the Table. In Figure 9 the computation time in second of the `LINSOLVEPALU_VEC` with $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$ and $B \in (\mathcal{M}_{d,1}(\mathbb{K}))^N$ for $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$ is represented.

Algorithm 42 Function `LINSOLVEPALU_VEC` , solves equation $\mathbf{A}\mathbf{X} = B$ where \mathbf{A} is a regular 3D-array (vectorized)

```

Function  $\mathbf{X} \leftarrow \text{LINSOLVEPALU\_VEC}(\mathbf{A}, B)$ 
   $[\mathbf{P}, \mathbf{L}, \mathbf{U}] \leftarrow \text{PALU\_VEC}(A)$ 
   $\mathbf{Y} \leftarrow \text{LINSOLVETRIL\_VEC}(\mathbf{L}, \text{MTIMES\_VEC}(\mathbf{P}, B))$ 
   $\mathbf{X} \leftarrow \text{LINSOLVETRIU\_VEC}(\mathbf{U}, \mathbf{Y})$ 
end Function

```

N	Matlab	Octave	Python	Python[Nat]
200 000	0.108(s)	0.111(s)	0.148(s)	0.036(s)
400 000	0.211(s)	0.212(s)	0.333(s)	0.071(s)
600 000	0.330(s)	0.334(s)	0.554(s)	0.107(s)
800 000	0.434(s)	0.473(s)	0.775(s)	0.143(s)
1 000 000	0.526(s)	0.614(s)	0.961(s)	0.180(s)
5 000 000	4.545(s)	4.823(s)	6.994(s)	0.922(s)
10 000 000	9.371(s)	9.963(s)	13.838(s)	1.854(s)

Table 27: Computational times in seconds of the `LINSOLVEPALU_VEC` function with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $B \in (\mathcal{M}_{3,1}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. The last column is for the native python function `numpy.linalg.solve`.

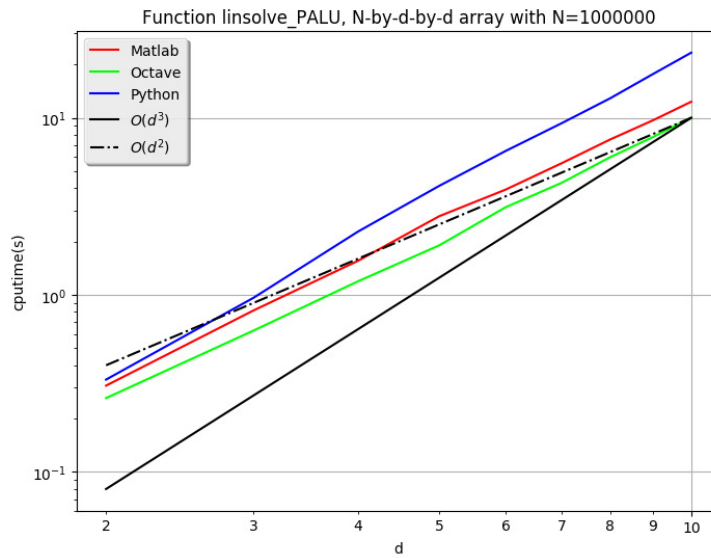


Figure 9: Computational times in seconds of `LINSOLVEPALU_VEC` function with $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$ with $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

6 Determinants

The purpose of this section is to compute determinant of an 3D-array as defined in section 1.2.1. In vectorized languages including a determinant function `DET` for a matrix in $\mathcal{M}_d(\mathbb{K})$, a non-vectorized code is easy to write and it is given in Algorithm 43. However, as N supposed to be very large compared to d we must vectorized determinants computation. It should be noted that the `numpy.linalg.det` Python function of the Numpy package can performed directly this operation.

Algorithm 43 Function `DET_MAT` , returns determinants of a 3D-array (not vectorized)

Input \mathbf{A} : in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$

Output \mathbf{D} : in \mathbb{K}^N

```

Function  $\mathbf{D} \leftarrow \text{DET\_MAT}(\mathbf{A})$ 
  for  $k \leftarrow 1$  to  $N$  do
     $\mathbf{D}(k) \leftarrow \text{DET}(\mathbf{A}(k, :, :))$ 
  end for
end Function

```

6.1 Vectorized algorithm using the Laplace expansion

To compute the determinant of a matrix $\mathbb{B} \in \mathcal{M}_n(\mathbb{R})$ we can use the Laplace expansion algorithm. For example the formula, expanded with respect to the i -th row is

$$\det \mathbb{B} = \sum_{j=1}^n b_{i,j} C_{i,j} \stackrel{\text{def}}{=} \det_1 \mathbb{B} \quad (24)$$

where the $C_{i,j}$ scalar is the (i, j) cofactor of \mathbb{B} . More precisely we have

$$C_{i,j} = (-1)^{i+j} M_{i,j}$$

where $M_{i,j}$ is the (i, j) minor of \mathbb{B} which is the determinant of the matrix formed by deleting the i -th row and the j -th columns of \mathbb{B} . We give in Algorithm 44 the **recursive** function `DETLAP` using the formula (24).

For a 3D-array \mathbf{A} in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$, we deduce the two non-vectorized function using the `DETLAP` function given in Algorithms 45 and 46. From the last one, we easily obtain the vectorized function `DETLAP_VEC` given in Algorithm 47.

Algorithm 44 Function `DET_LAP`, returns determinant of the matrix \mathbb{B} by using Laplace formula (24) expanded with respect to the 1-st row.

Input \mathbb{B} : a d -by- d matrix

Output r : the scalar $\det(\mathbb{B})$.

```

Function  $r \leftarrow \text{DET\_LAP}(\mathbb{B})$ 
  if  $d == 1$  then
     $r \leftarrow \mathbb{B}(1, 1)$ 
  else
     $r \leftarrow 0$ 
    for  $j \leftarrow 1$  to  $d$  do
       $r \leftarrow r + (-1)^{1+j} * \mathbb{B}(1, j) *$ 
         $\text{DET\_LAP}(\mathbb{B}(2 : d, [1 : j - 1, j + 1 : d]))$ 
    end for
  end if
end Function

```

Algorithm 46 Function `DET_LAP_CPT`, returns determinants of a 3D-array in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$ (not vectorized)

```

Function  $D \leftarrow \text{DET\_LAP\_CPT}(\mathbb{A})$ 
  if  $d == 1$  then
     $D \leftarrow \mathbb{A}$ 
  else
     $D \leftarrow \text{ZEROS}(N, 1)$ 
    for  $k \leftarrow 1$  to  $N$  do
      for  $j \leftarrow 1$  to  $d$  do
         $D(k) \leftarrow D(k) + (-1)^{j+1} * \mathbb{A}(k, 1, j) *$ 
           $\text{DET\_LAP}(\mathbb{A}(k, 2 : d, [1 : j - 1, j + 1 : d]))$ 
      end for
    end for
  end if
end Function

```

Algorithm 45 Function `DET_LAP_MAT`, returns determinants of a 3D-array (not vectorized)

```

Function  $D \leftarrow \text{DET\_LAP\_MAT}(\mathbb{A})$ 
  for  $k \leftarrow 1$  to  $N$  do
     $D(k) \leftarrow \text{DET\_LAP}(\mathbb{A}(k, :, :))$ 
  end for
end Function

```

Algorithm 47 Function `DET_LAP_VEC`, returns determinants of a 3D-array in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$ (not vectorized)

```

Function  $D \leftarrow \text{DET\_LAP\_VEC}(\mathbb{A})$ 
  if  $d == 1$  then
     $D \leftarrow \mathbb{A}$ 
  else
     $D \leftarrow \text{ZEROS}(N, 1)$ 
    for  $j \leftarrow 1$  to  $d$  do
       $D \leftarrow D + (-1)^{j+1} * \mathbb{A}(:, 1, j) *$ 
         $\text{DET\_LAP\_VEC}(\mathbb{A}(:, 2 : d, [1 : j - 1, j + 1 : d]))$ 
    end for
  end if
end Function

```

The major disadvantage of the Algorithm 47 is that it is memory consuming. To overcome, instead of creating a new 3D array from \mathbb{A} when calling recursively the function, we only create a row and column indices as 1D arrays. This is the object of the Algorithm 48.

Algorithm 48 Function `DET_LAP_IDX`, returns determinants of \mathbb{A}_k matrices by using Laplace formula (24) expanded with respect to the 1-st row (vectorized, recursive and memory safe).

Input \mathbb{A} : in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$
 I : (optional) row indices. default $1 : d$.
 Always the same size as J .
 J : (optional) column indices. default $1 : d$.
 Always the same size as I .

Output D : in \mathbb{K}^N

```

Function  $D \leftarrow \text{DET\_LAP\_IDX}(\mathbb{A}, I, J)$ 
  if  $I = \emptyset$  and  $J = \emptyset$  then
     $m \leftarrow d$ 
     $I \leftarrow 1 : d, J \leftarrow 1 : d$ 
  else
     $m \leftarrow \text{LEN}(I)$ 
  end if
  if  $m == 1$  then
     $D \leftarrow \mathbb{A}(:, I(1), J(1))$ 
  else
     $D \leftarrow \text{ZEROS}(1, N)$ 
    for  $j \leftarrow 1$  to  $m$  do
       $D \leftarrow D + (-1)^{1+j} * \mathbb{A}(:, I(1), J(j)) *$ 
         $\text{DET\_LAP\_IDX}(\mathbb{A}, I(2 : m), J([1 : j - 1, j + 1 : m]))$ 
    end for
  end if
end Function

```

N	Matlab	Octave	Python	N	Matlab	Octave	Python
200 000	6.723(s)	60.282(s)	38.974(s)	200 000	5.538(s)	59.979(s)	39.098(s)
400 000	13.522(s)	121.649(s)	78.228(s)	400 000	11.078(s)	120.601(s)	78.053(s)
600 000	20.281(s)	181.437(s)	117.235(s)	600 000	16.642(s)	181.110(s)	119.746(s)
800 000	27.172(s)	239.521(s)	156.981(s)	800 000	22.174(s)	238.726(s)	155.709(s)
1 000 000	35.669(s)	300.370(s)	196.141(s)	1 000 000	29.232(s)	298.881(s)	195.936(s)

(a) Function <code>DETLAP_CPT</code>				(b) Function <code>DETLAP_MAT</code>			
N	Matlab	Octave	Python	N	Matlab	Octave	Python
200 000	0.009(s)	0.007(s)	0.018(s)	200 000	0.006(s)	0.004(s)	0.008(s)
400 000	0.016(s)	0.015(s)	0.038(s)	400 000	0.010(s)	0.009(s)	0.020(s)
600 000	0.024(s)	0.023(s)	0.058(s)	600 000	0.014(s)	0.015(s)	0.032(s)
800 000	0.033(s)	0.031(s)	0.078(s)	800 000	0.019(s)	0.021(s)	0.045(s)
1 000 000	0.043(s)	0.058(s)	0.152(s)	1 000 000	0.024(s)	0.027(s)	0.056(s)
5 000 000	0.428(s)	0.549(s)	0.826(s)	5 000 000	0.246(s)	0.319(s)	0.398(s)
10 000 000	0.842(s)	1.067(s)	1.651(s)	10 000 000	0.490(s)	0.634(s)	0.796(s)

(c) Function <code>DETLAP_VEC</code>				(d) Function <code>DETLAPIDX_VEC</code>			
N	Matlab	Matlab(*)	Octave	Python	Python(Nat)		
200 000	0.009(s)	0.009(s)	0.007(s)	0.018(s)	0.040(s)		
400 000	0.016(s)	0.018(s)	0.015(s)	0.038(s)	0.081(s)		
600 000	0.024(s)	0.028(s)	0.023(s)	0.058(s)	0.121(s)		
800 000	0.033(s)	0.038(s)	0.031(s)	0.078(s)	0.161(s)		
1 000 000	0.043(s)	0.049(s)	0.058(s)	0.152(s)	0.201(s)		
5 000 000	0.428(s)	0.492(s)	0.549(s)	0.826(s)	0.999(s)		
10 000 000	0.842(s)	0.972(s)	1.067(s)	1.651(s)	2.041(s)		

Table 28: Computational times in seconds of `DETLAPIDX` functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

N	Matlab	Matlab(*)	Octave	Python	Python(Nat)
200 000	0.009(s)	0.009(s)	0.007(s)	0.018(s)	0.040(s)
400 000	0.016(s)	0.018(s)	0.015(s)	0.038(s)	0.081(s)
600 000	0.024(s)	0.028(s)	0.023(s)	0.058(s)	0.121(s)
800 000	0.033(s)	0.038(s)	0.031(s)	0.078(s)	0.161(s)
1 000 000	0.043(s)	0.049(s)	0.058(s)	0.152(s)	0.201(s)
5 000 000	0.428(s)	0.492(s)	0.549(s)	0.826(s)	0.999(s)
10 000 000	0.842(s)	0.972(s)	1.067(s)	1.651(s)	2.041(s)

Table 29: Computational times in seconds of `DETLAP_VEC` functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. The last column is for the native python function `numpy.linalg.det`.

N	Matlab	Matlab(*)	Octave	Python	Python(Nat)
200 000	0.006(s)	0.006(s)	0.004(s)	0.008(s)	0.040(s)
400 000	0.010(s)	0.011(s)	0.009(s)	0.020(s)	0.081(s)
600 000	0.014(s)	0.017(s)	0.015(s)	0.032(s)	0.121(s)
800 000	0.019(s)	0.024(s)	0.021(s)	0.045(s)	0.161(s)
1 000 000	0.024(s)	0.031(s)	0.027(s)	0.056(s)	0.201(s)
5 000 000	0.246(s)	0.305(s)	0.319(s)	0.398(s)	0.999(s)
10 000 000	0.490(s)	0.606(s)	0.634(s)	0.796(s)	2.041(s)

Table 30: Computational times in seconds of `DETLAPIDX_VEC` functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. The last column is for the native python function `numpy.linalg.det`.

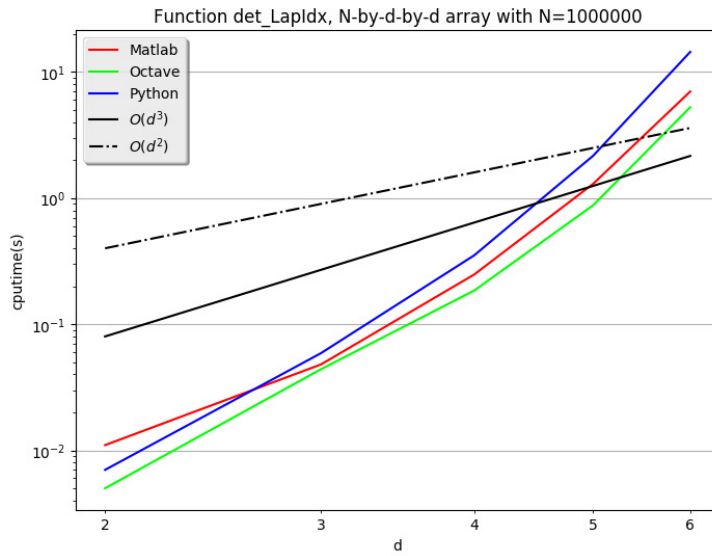


Figure 10: Computational times in seconds of `DET_LAP_IDX_VEC` function with $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$ with $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

6.2 Using LU factorization

An other way in calculating determinant of a matrix is to use the LU factorization with partial pivoting. Indeed we have

$$\mathbb{P}\mathbb{A} = \mathbb{L}\mathbb{U}. \quad (25)$$

where $\mathbb{P} \in \mathcal{M}_d(\mathbb{R})$ is a permutation matrix, $\mathbb{L} \in \mathcal{M}_d(\mathbb{C})$ is a lower triangular matrix with unit diagonal and $\mathbb{U} \in \mathcal{M}_d(\mathbb{C})$ is an upper triangular matrix. So we obtain

$$\det \mathbb{P} \det \mathbb{A} = \det \mathbb{L} \det \mathbb{U}$$

As \mathbb{P} is a permutation matrix we have

$$\det \mathbb{P} = \pm 1$$

and we The parity (oddness or evenness) of a permutation σ of $\llbracket 1, d \rrbracket$ can be defined as the parity of the number of inversions for σ , i.e., of pairs of elements i, j of $\llbracket 1, d \rrbracket$ such that $i < j$ and $\sigma(i) > \sigma(j)$. The sign or signature of a permutation σ is denoted $\text{sign } \sigma$ and defined as $+1$ if σ is even and -1 if σ is odd.

N	Matlab	Matlab(*)	Octave	Python	Python(Nat)
200 000	0.006(s)	0.006(s)	0.004(s)	0.085(s)	0.040(s)
400 000	0.009(s)	0.010(s)	0.008(s)	0.186(s)	0.081(s)
600 000	0.014(s)	0.016(s)	0.013(s)	0.314(s)	0.121(s)
800 000	0.018(s)	0.022(s)	0.018(s)	0.455(s)	0.161(s)
1 000 000	0.023(s)	0.029(s)	0.022(s)	0.574(s)	0.201(s)
5 000 000	0.226(s)	0.297(s)	0.272(s)	3.797(s)	0.999(s)
10 000 000	0.452(s)	0.597(s)	0.542(s)	7.544(s)	2.041(s)

Table 31: Computational times in seconds of `DETPLUIN_VEC` function with $\mathbb{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. The last column is for the native python function `numpy.linalg.det`.

6.3 Vectorized algorithm using an other expansion

In [3], an other expansion to compute the determinant of a matrix $\mathbb{B} \in \mathcal{M}_n(\mathbb{R})$ is given by

$$\det \mathbb{B} = \frac{\det(\mathbb{M}^{[1,1]}) \det(\mathbb{M}^{[n,n]}) - \det(\mathbb{M}^{[n,1]}) \det(\mathbb{M}^{[1,n]})}{\det(\mathbb{B}^{[2]})} \stackrel{\text{def}}{=} \det_2 \mathbb{B} \quad (26)$$

where $\mathbb{M}^{[i,j]}$ is the matrix formed by deleting the i -th row and the j -th columns of \mathbb{B} and $\mathbb{B}^{[2]}$ is the submatrix of \mathbb{B} formed by deleting rows $1, n$ and columns $1, n$ of \mathbb{B} . This formula is not really usefull as a divide by zero is always possible even if the matrix \mathbb{B} is symmetric positive definite. For example for any symmetric positive definite matrix $\mathbb{B} \in \mathcal{M}_4(\mathbb{R})$ such that $\mathbb{B}_{2,3} = \mathbb{B}_{3,2} = 0$ a division by zero occurs in computation of $\det(\mathbb{M}^{[n,1]})$. Try the identity matrix!

However when the matrix \mathbb{B} is symmetric positive definite one can mixed formulas (24) and (26) to obtain

$$\det \mathbb{B} = \frac{\det_2(\mathbb{M}^{[1,1]}) \det_2(\mathbb{M}^{[n,n]}) - \det_1(\mathbb{M}^{[n,1]})^2}{\det_2(\mathbb{B}^{[2]})} \stackrel{\text{def}}{=} \det_3 \mathbb{B} \quad (27)$$

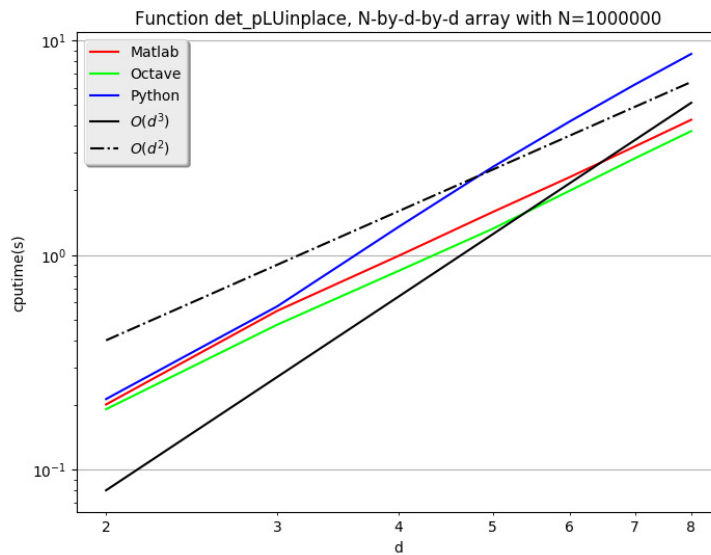


Figure 11: Computational times in seconds of `DETPLUIN_VEC` function with $\mathbf{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$ with $N = 10^6$ and $d \in \llbracket 2, 10 \rrbracket$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

It is clear that the matrices $\mathbb{M}^{[1,1]}$, $\mathbb{M}^{[n,n]}$ and $\mathbb{B}^{[2]}$ are also symmetric positive definite. Due to symmetry we have $\mathbb{M}^{[n,1]} = \mathbb{M}^{[1,n]}$ but this matrix is not symmetric positive definite and so if we apply formula (26) on it a division by zero may occur. To overcome this problem we used formula (24) to compute the determinant of $\mathbb{M}^{[n,1]}$.

We give in Algorithm 49 the **recursive** function using the formula (27).

Algorithm 49 Function `DETMIXED`, returns determinant of the symmetric/hermitian positive definite matrix \mathbb{B} by using formula (27).

Input \mathbb{B} : a n -by- n matrix

Output r : the scalar $\det(\mathbb{B})$.

```

Function  $r \leftarrow$  DETMIXED ( $\mathbb{B}$ )
  if  $n == 1$  then
     $r \leftarrow \mathbb{B}(1, 1)$ 
  else if  $n == 2$  then
     $r \leftarrow \mathbb{B}(1, 1) * \mathbb{B}(2, 2) - \mathbb{B}(1, 2) * \mathbb{B}(2, 1)$ 
  else
     $r \leftarrow$  DETMIXED ( $\mathbb{B}(2 : n, 2 : n)$ ) * DETMIXED ( $\mathbb{B}(1 : n - 1, 1 : n - 1)$ )
     $r \leftarrow r -$  DETLAPLACE( $\mathbb{B}(1 : n - 1, 2 : n)$ )2
     $r \leftarrow r /$  DETMIXED ( $\mathbb{B}(2 : n - 1, 2 : n - 1)$ )
  end if
end Function

```

A first vectorized and recursive function using the 3D array \mathbb{A} is easy to write and it is given in Algorithm 50

Algorithm 50 Function `DETMIX_VEC`, returns determinants of symmetric/hermitian positive definite matrices \mathbb{A}_k by using formula (27) (vectorized and recursive).

Input \mathbb{A} : N -by- d -by- d 3D array such that
 $\mathbb{A}(k, :, :) = \mathbb{A}_k, \forall k \in \llbracket 1, N \rrbracket$.

Output \mathbf{D} : array of size N such that
 $\mathbf{D}(k) = \det(\mathbb{A}_k), \forall k \in \llbracket 1, N \rrbracket$.

```

Function  $\mathbf{D} \leftarrow$  DETMIX_VEC ( $\mathbb{A}$ )
  if  $d == 1$  then
     $\mathbf{D} \leftarrow \mathbb{A}(:, 1, 1)$ 
  else if  $d == 2$  then
     $\mathbf{D} \leftarrow \mathbb{A}(:, 1, 1) .* \mathbb{A}(:, 2, 2) - \mathbb{A}(:, 1, 2) .* \mathbb{A}(:, 2, 1)$ 
  else
     $\mathbf{I}_1 \leftarrow 2 : d, \mathbf{I}_d \leftarrow 1 : d - 1, \mathbf{I}_{1d} \leftarrow 2 : d - 1$ 
     $\mathbf{D} \leftarrow$  DETMIX_VEC( $\mathbb{A}(:, \mathbf{I}_1, \mathbf{I}_1)$ ) .* DETMIX_VEC( $\mathbb{A}(:, \mathbf{I}_d, \mathbf{I}_d)$ )
     $\mathbf{D} \leftarrow \mathbf{D} -$  DETLAP_VEC( $\mathbb{A}(:, \mathbf{I}_1, \mathbf{I}_d)$ )2
     $\mathbf{D} \leftarrow \mathbf{D} ./$  DETMIX_VEC( $\mathbb{A}(:, \mathbf{I}_{1d}, \mathbf{I}_{1d})$ )
  end if
end Function

```

The major disadvantage of the Algorithm 50 is that it is memory consuming.

To overcome, instead of creating a new 3D array from \mathbb{A} when calling recursively the function, we only create a row and column indices as 1D arrays. This is the object of the Algorithm 51.

Algorithm 51 Function `DETVEC_v04`, returns determinants of the symmetric definite positive matrices \mathbb{A}_k by using formula (26) (vectorized, recursive and memory safe).

Input \mathbb{A} : N -by- d -by- d 3D array such that
 $\mathbb{A}(k, :, :) = \mathbb{A}_k, \forall k \in \llbracket 1, N \rrbracket$.
 \mathbf{I} : (optional) row indices. default $1 : d$.
Always the same size as \mathbf{J} .
 \mathbf{J} : (optional) column indices. default $1 : d$.
Always the same size as \mathbf{I} .

Output \mathbf{D} : array of size N such that
 $\mathbf{D}(k) = \det(\mathbb{A}_k), \forall k \in \llbracket 1, N \rrbracket$.

```

Function  $\mathbf{D} \leftarrow \text{DETMIXIDX\_VEC}(\mathbb{A}\{\mathbf{I}, \mathbf{J}\})$ 
  if  $\mathbf{I} = \emptyset$  and  $\mathbf{J} = \emptyset$  then
     $m \leftarrow d$ 
     $\mathbf{I} \leftarrow 1 : d, \mathbf{J} \leftarrow 1 : d$ 
  else
     $m \leftarrow \text{LEN}(\mathbf{I})$ 
  end if
  if  $m == 1$  then
     $\mathbf{D} \leftarrow \mathbb{A}(:, 1, 1)$ 
  else if  $m == 2$  then
     $\mathbf{D} \leftarrow \mathbb{A}(:, 1, 1) .* \mathbb{A}(:, 2, 2) - \mathbb{A}(:, 1, 2) .* \mathbb{A}(:, 2, 1)$ 
  else
     $\mathbf{I}_1 \leftarrow \mathbf{I}(2 : m), \mathbf{J}_1 \leftarrow \mathbf{J}(2 : m),$ 
     $\mathbf{I}_d \leftarrow \mathbf{I}(1 : m - 1), \mathbf{J}_d \leftarrow \mathbf{J}(1 : m - 1),$ 
     $\mathbf{I}_{1d} \leftarrow \mathbf{I}(2 : m - 1), \mathbf{J}_{1d} \leftarrow \mathbf{J}(2 : m - 1),$ 
     $\mathbf{D} \leftarrow \text{DETMIXIDX\_VEC}(\mathbb{A}, \mathbf{I}_1, \mathbf{J}_1) .* \text{DETMIXIDX\_VEC}(\mathbb{A}, \mathbf{I}_d, \mathbf{I}_d)$ 
     $\mathbf{D} \leftarrow \mathbf{D} - \text{DETLAPIDX\_VEC}(\mathbb{A}, \mathbf{I}_1, \mathbf{I}_d) .^2$ 
     $\mathbf{D} \leftarrow \mathbf{D} ./ \text{DETMIXIDX\_VEC}(\mathbb{A}, \mathbf{I}_{1d}, \mathbf{I}_{1d})$ 
  end if
end Function

```

N	Matlab	Octave	Python	N	Matlab	Octave	Python
200 000	0.013(s)	0.011(s)	0.012(s)	200 000	0.008(s)	0.006(s)	0.009(s)
400 000	0.025(s)	0.026(s)	0.024(s)	400 000	0.014(s)	0.014(s)	0.018(s)
600 000	0.043(s)	0.033(s)	0.036(s)	600 000	0.025(s)	0.024(s)	0.027(s)
800 000	0.061(s)	0.046(s)	0.048(s)	800 000	0.036(s)	0.028(s)	0.036(s)
1 000 000	0.076(s)	0.048(s)	0.060(s)	1 000 000	0.045(s)	0.032(s)	0.045(s)
5 000 000	0.243(s)	0.239(s)	0.329(s)	5 000 000	0.160(s)	0.154(s)	0.249(s)
10 000 000	0.477(s)	0.464(s)	0.650(s)	10 000 000	0.313(s)	0.295(s)	0.498(s)

(a) Function `DETMIX_VEC`

(b) Function `DETMIXIDX_VEC`

Table 32: Computational times in seconds of `DETMIX` functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

A Vectorized algorithmic language

A.1 Common operators and functions

We also provide below some common functions and operators of the vectorized algorithmic language used in this article which generalize the operations on scalars to higher dimensional arrays, matrices and vectors:

$\mathbb{A} \leftarrow \mathbb{B}$	Assignment
$\mathbb{A} * \mathbb{B}$	matrix multiplication,
$\mathbb{A} .* \mathbb{B}$	element-wise multiplication,
$\mathbb{A} ./ \mathbb{B}$	element-wise division,
$\mathbb{A}(:)$	all the elements of \mathbb{A} , regarded as a single column.
\mathbb{A}^t	transpose of the matrix \mathbb{A} .
$[,]$	Horizontal concatenation,
$[:,]$	Vertical concatenation,
$\mathbb{A}(:, J)$	J -th column of \mathbb{A} ,
$\mathbb{A}(I, :)$	I -th row of \mathbb{A} ,
$\text{SUM}(\mathbb{A}, dim)$	sums along dimension dim ,
$\text{PROD}(\mathbb{A}, dim)$	product along dimension dim ,
$\text{SIZE}(\mathbb{A})$	return the dimensions of the multi-array \mathbb{A}
$\text{ARGMAX}(\mathbb{A}, dim)$	the arguments of the maxima along dimension dim ,
$\text{ARGMIN}(\mathbb{A}, dim)$	the arguments of the minimum along dimension dim ,
\mathbb{I}_n	n -by- n identity matrix,
$\mathbb{1}_{m \times n}$ (or $\mathbb{1}_n$)	m -by- n (or n -by- n) matrix or sparse matrix of ones,
$\mathbb{0}_{m \times n}$ (or $\mathbb{0}_n$)	m -by- n (or n -by- n) matrix or sparse matrix of zeros,
$\text{ONES}(n_1, n_2, \dots, n_\ell)$	ℓ dimensional array of ones,
$\text{ZEROS}(n_1, n_2, \dots, n_\ell)$	ℓ dimensional array of zeros,
$\text{REPTILE}(\mathbb{A}, m, n)$	tiles the p -by- q array/matrix \mathbb{A} to produce the $(m \times p)$ -by- $(n \times q)$ array composed of copies of \mathbb{A} ,
$\text{RESHAPE}(\mathbb{A}, m, n)$	returns the m -by- n array/matrix whose elements are taken columnwise from \mathbb{A} .
$\text{DET}(\mathbb{A})$	return the determinant of the square matrix \mathbb{A} .
$\text{SUB2IND}(dims, i_1, i_2, \dots, i_d)$	return the linear index corresponding to the provided subscripts of an array of dimensions $dims$. Here d is the number of dimensions i.e. the length of the $dims$ array.
$\text{IND2SUB}(dims, index)$	return the subscripts of corresponding to the provided linear index of an array of dimension $dims$

In vectorized language *broadcasting* provides a means of vectorizing array operations so that looping occurs in low level language as C, Fortran. Element-wise operations between two multi-dimensionnals arrays are said to be compatible if the smaller array is *broadcast* across the larger array so that they have compatible dimensions. Let A be a n_1 -by- n_2 -by-...-by- n_a array and B be a m_1 -by- m_2 -by-...-by- m_b with $n_a \leq m_b$. These two arrays are compatible for element wise operations if

$$n_i = m_i \text{ or } n_i = 1 \text{ or } m_i = 1, \quad \forall i \in \llbracket 1, n_a \rrbracket$$

Let \square denote an element wise binary operator. If the two arrays A and B are compatible then the following operations are allowed

$$C \leftarrow A \square B \quad \text{and} \quad D \leftarrow B \square A.$$

The result arrays C and D have the same dimension
 $\max(n_1, m_1)$ -by- $\max(n_2, m_2)$ -by-...-by- $\max(n_a, m_a)$ -by- m_{a+1} -by-...-by- m_b

A.1.1 SUB2IND function

$I \leftarrow \text{SUB2IND}([d_1, \dots, d_n], i_1, \dots, i_n)$ returns the linear index I corresponding to the provided subscripts i_1, \dots, i_n of an n multi-dimensionnal array of dimensions $[d_1, \dots, d_n]$. Subscripts i_1, \dots, i_n must have the same size and index I will have this size. For example if \mathbf{A} is a n -dimensional array and all subscripts i_1, \dots, i_n are 1-dimensionnal array of dimension m then

$$I \leftarrow \text{SUB2IND}(\text{SIZE}(\mathbf{A}), i_1, \dots, i_n)$$

returns the linear index I which is the 1-dimensionnal array of dimension m such that

$$\mathbf{A}(I(k)) = \mathbf{A}(i_1(k), \dots, i_n(k)), \quad \forall k \in \llbracket 1, m \rrbracket$$

where $\mathbf{A}(I(k))$ is equivalent to $\mathbf{B}(I(k))$ where $\mathbf{B} = \mathbf{A}(\cdot)$.

A.1.2 IND2SUB function

A.2 Combinatorial functions

PERMS(\mathbf{V}) where \mathbf{V} is an array of length n . Returns a $n!$ -by- n array containing all permutations of \mathbf{V} elements.

The lexicographical order is chosen.

COMBS(\mathbf{V}, k) where \mathbf{V} is an array of length n and $k \in \llbracket 1, n \rrbracket$.

Returns a $\frac{n!}{k!(n-k)!}$ -by- k array containing all combinations of n elements taken k at a time. The lexicographical order is chosen.

B Information for developpers

git informations on the \LaTeX repository of this report

```
name: LinAlg3D
tag:
commit: c39bf1fcb6f0eec0282e90fa5601e9875dd2932
date: 2018-05-29
time: 10-43-01
status: True
```

git informations on the \LaTeX package used to build this report

```
name: fctools
tag:
commit: 72693985daa7d84c61906a71c61d15f33893c3f6
date: 2018-05-09
time: 13-36-42
status: True
```

```

git informations on the Matlab Tooboxes/Octave packages used to build this report
-----
name : fc-linalg3D
tag : 0.0.2
commit : 438689a4fc87c1771cf6d632799ab114f5767144
date : 2018-05-21
time : 09-11-47
status : 0
-----
name : fc-bench
tag : 0.0.5
commit : e83053f02f34ae036fd4e13ef68a50783b88d7b8
date : 2018-05-21
time : 09-01-13
status : 0
-----
name : fc-tools
tag : 0.0.23
commit : 5728a827d9e6b883bb8ba8005a83a1a3f7d16be8
date : 2018-05-14
time : 14-32-51
status : 0
-----

```

```

git informations on the Python packages used to build this report
-----
name: fc-linalg3D
tag: 0.0.1
commit: 4695121824be1b27da7ab127b50f7fdcad423d3a
date: 2018-05-20
time: 06-48-40
status: 0

name: fc-bench
tag: 0.0.3
commit: 6fabfb9ab5d08281670bb13131980bdecb58012
date: 2018-05-18
time: 12-55-12
status: 0

name: fc-tools
tag: 0.0.17
commit: 54bd33dbaeca9dfbd3efe73516b89840a6cb9bfe
date: 2018-05-20
time: 06-46-10
status: 0
-----

```

List of algorithms

1	Function <code>AXPBY_CPT</code> , returns linear combination $\alpha X + \beta Y$ by using component by component computation.	8
2	Function <code>GETCPT</code> , returns component (i, j) of the k -th <i>matrix</i> of X	8
3	Function <code>AXPBY_CVT</code> , returns linear combination $\alpha X + \beta Y$ by converting arrays to a 3D-arrays.	9
4	Function <code>TO3DARRAY</code> , converts to a 3Darray	9
5	Function <code>AXPBY_MAT</code> , returns linear combination $\alpha X + \beta Y$ by using vectorized operations on 2D-arrays or matrices.	9
6	Function <code>GETMAT</code> , returns the k -th <i>matrix</i> of X	9
7	Function <code>AXPBY_VEC</code> , returns linear combination $\alpha X + \beta Y$ by using vectorized operations on 1D-arrays.	9
8	Function <code>GETVEC</code> , returns (i, j) components of X	9

9	Function EBYE_CPT , returns element by element operation $X \diamond Y$. Here f is the function $f : (x, y) \in \mathbb{K}^2 \longrightarrow x \diamond y$	12
10	Function EBYE_MAT , returns element by element operation $X \diamond Y$ by using function $f : (\mathbb{A}, \mathbb{B}) \longrightarrow \mathbb{A} \diamond \mathbb{B}$ where \mathbb{A} and \mathbb{B} are in $\mathcal{M}_{m,n}(\mathbb{K})$ or in \mathbb{K}	12
11	Function EBYE_VEC , returns element by element operation $X \diamond Y$ by using function $f : (\mathbf{A}, \mathbf{B}) \longrightarrow \mathbf{A} \diamond \mathbf{B}$ where \mathbf{A} and \mathbf{B} are in \mathbb{K}^N	12
12	Function EBYE_CVT , returns element by element operation $X \diamond Y$ by converting arrays to a 3D-arrays. Here f is the function $f : (\mathbf{A}, \mathbf{B}) \longrightarrow \mathbf{A} \diamond \mathbf{B}$ where \mathbf{A} and \mathbf{B} are in $(\mathcal{M}_{m,n}(\mathbb{K}))^N$	12
13	Function TIMES_CPT , returns element by element product $X.*Y$	13
14	Function TIMES_MAT , returns element by element product $X.*Y$ by using vectorized operations on 2D-arrays or matrices.	13
15	Function TIMES_VEC , returns element by element product $X.*Y$ by using vectorized operations on 1D-arrays.	13
16	Function TIMES_CVT , returns element by element product $X.*Y$ by converting arrays to a 3D-arrays.	13
17	Function MTIMES_CPT , returns matricial products $X * Y$ where X or/and Y are 3D-arrays.	14
18	Function MTIMES_VEC , returns matricial products $X * Y$ where X or/and Y are 3D-arrays.	14
19	Function MTIMES_MAT , returns matricial products $X * Y$ where X or/and Y are 3D-arrays.	14
20	Function LINSOLVEDIAG , solves diagonal linear system $\mathbb{A}\mathbb{X} = \mathbb{B}$	18
21	Function LINSOLVETriL . Returns solution of equation $\mathbb{A}\mathbb{X} = \mathbb{B}$ where \mathbf{A} is a regular lower triangular matrix.	19
22	Function LINSOLVETriL_MAT , solves equation $\mathbf{A}\mathbb{X} = B$ where \mathbf{A} is a regular lower triangular 3D-array(not vectorized)	19
23	Function LINSOLVETriL_CPT , solves equation $\mathbf{A}\mathbb{X} = B$ where \mathbf{A} is a regular lower triangular 3D-array(not vectorized)	19
24	Function LINSOLVETriL_VEC , solves equation $\mathbf{A}\mathbb{X} = B$ where \mathbf{A} is a regular lower triangular 3D-array(vectorized)	19
25	Function LINSOLVETriU . Returns solution of equation $\mathbb{A}\mathbb{X} = \mathbb{B}$ where \mathbf{A} is a regular upper triangular matrix.	22
26	Function LINSOLVETriU_MAT , solves equation $\mathbf{A}\mathbb{X} = B$ where \mathbf{A} is a regular upper triangular 3D-array(not vectorized)	22
27	Function LINSOLVETriU_CPT , solves equation $\mathbf{A}\mathbb{X} = B$ where \mathbf{A} is a regular upper triangular 3D-array(not vectorized)	22
28	Function LINSOLVETriU_VEC , solves equation $\mathbf{A}\mathbb{X} = B$ where \mathbf{A} is a regular upper triangular 3D-array(vectorized)	22
29	Function CHOLESKY . Computes the lower triangular matrix $\mathbb{L} \in \mathcal{M}_n(\mathbb{C})$ such that $\mathbb{B} = \mathbb{L}\mathbb{L}^*$	26
30	Function CHOLESKY_MAT , returns cholesky factorizations of A_k matrices (not vectorized)	27
31	Function CHOLESKY_CPT , returns cholesky factorizations of \mathbf{A}_k matrices (not vectorized)	27
32	Function CHOLESKY_VEC , returns cholesky factorizations of A_k matrices (vectorized)	28

33	Function PALU computes the LU factorization with partial pivoting of a matrix \mathbf{A} such that $\mathbb{P}\mathbf{A} = \mathbb{L}\mathbf{U}$	31
34	Function PLUINPLACE inplace computation of the LU factorization with partial pivoting of a matrix \mathbf{A} such that $\mathbb{P}\mathbf{A} = \mathbb{L}\mathbf{U}$	32
35	Function PALU_MAT computes all LU factorizations with partial pivoting of a 3D-array \mathbf{A} such that $\mathbb{P}_k\mathbf{A}_k = \mathbb{L}_k\mathbf{U}_k$	32
36	Function PALU_CPT computes all LU factorizations with partial pivoting of a 3D-array \mathbf{A} such that $\mathbb{P}_k\mathbf{A}_k = \mathbb{L}_k\mathbf{U}_k$	33
37	Function PALU_VEC computes all LU factorizations with partial pivoting of a 3D-array \mathbf{A} such that $\mathbb{P}_k\mathbf{A}_k = \mathbb{L}_k\mathbf{U}_k$	34
38	Function PLUINPLACE_MAT computes all LU factorizations with partial pivoting of a 3D-array \mathbf{A} such that $\mathbb{P}_k\mathbf{A}_k = \mathbb{L}_k\mathbf{U}_k$	36
39	Function PLUINPLACE_CPT computes all LU factorizations with partial pivoting of a 3D-array \mathbf{A} such that $\mathbb{P}_k\mathbf{A}_k = \mathbb{L}_k\mathbf{U}_k$	37
40	Function PLUINPLACE_VEC computes all LU factorizations with partial pivoting of a 3D-array \mathbf{A} such that $\mathbb{P}_k\mathbf{A}_k = \mathbb{L}_k\mathbf{U}_k$	37
41	Function LINSOLVECHOLESKY_VEC , solves equation $\mathbf{A}\mathbf{X} = \mathbf{B}$ where \mathbf{A} is a symmetric positive definite 3D-array(vectorized)	40
42	Function LINSOLVEPALU_VEC , solves equation $\mathbf{A}\mathbf{X} = \mathbf{B}$ where \mathbf{A} is a regular 3D-array (vectorized)	42
43	Function DET_MAT , returns determinants of a 3D-array (not vectorized)	44
44	Function DETLAP , returns determinant of the matrix \mathbb{B} by using Laplace formula (24) expanded with respect to the 1-st row.	45
45	Function DETLAP_MAT , returns determinants of a 3D-array (not vectorized)	45
46	Function DETLAP_CPT , returns determinants of a 3D-array in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$ (not vectorized)	45
47	Function DETLAP_VEC , returns determinants of a 3D-array in $(\mathcal{M}_{d,d}(\mathbb{K}))^N$ (not vectorized)	45
48	Function DETLAPIDX , returns determinants of \mathbf{A}_k matrices by using Laplace formula (24) expanded with respect to the 1-st row (vectorized, recursive and memory safe).	45
49	Function DETMIXED , returns determinant of the symmetric/hermitian positive definite matrix \mathbb{B} by using formula (27).	50
50	Function DETMIX_VEC , returns determinants of symmetric/hermitian positive definite matrices \mathbf{A}_k by using formula (27) (vectorized and recursive).	50
51	Function DETVEC_v04 , returns determinants of the symmetric definite positive matrices \mathbf{A}_k by using formula (26) (vectorized, recursive and memory safe).	51

List of Tables

1	Common element by element operations	5
2	Computational times in seconds of AXPBV functions with $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	10

3	Computational times in seconds of <code>AXPBY_NAT</code> functions with X in $(\mathcal{M}_{3,3}(\mathbb{K}))^N$ and Y in $\mathcal{M}_{3,3}(\mathbb{R})$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. Matlab(*) refers to Matlab without multi-threadings.	11
4	Computational times in seconds of <code>AXPBY_NAT</code> functions with X and Y in $(\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. Matlab(*) refers to Matlab without multi-threadings.	11
5	Function <code>AXPBY_MAT</code> with $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$ under Matlab 2018a: effects of multithreading on cputimes . . .	11
6	Function <code>AXPBY_VEC</code> with $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$ under Matlab 2018a: effects of multithreading on cputimes . . .	11
7	Function <code>AXPBY_CVT</code> with $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$ under Matlab 2018a: effects of multithreading on cputimes . . .	12
8	Function <code>AXPBY_NAT</code> with $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$ under Matlab 2018a: effects of multithreading on cputimes . . .	12
9	Computational times in seconds of <code>TIMES</code> functions with $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	13
10	Computational times in seconds of <code>MTIMES</code> functions with $X \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $Y \in \mathcal{M}_{3,3}(\mathbb{R})$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	15
11	Computational times in seconds of <code>MTIMES_VEC</code> functions with X in $(\mathcal{M}_{3,3}(\mathbb{K}))^N$ and Y in $\mathcal{M}_{3,3}(\mathbb{R})$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. Matlab(*) refers to Matlab without multi-threadings and Python(Nat) to Numpy <code>matmul</code> function. . .	15
12	Computational times in seconds of <code>MTIMES_VEC</code> functions with X and Y in $(\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. Matlab(*) refers to Matlab without multi-threadings and Python(Nat) to Numpy <code>matmul</code> function.	15
13	Computational times in seconds of <code>LINSOLVETRI_L</code> functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	20
14	Computational times in seconds of the <code>LINSOLVETRI_L_VEC</code> function with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	20
15	Computational times in seconds of <code>LINSOLVETRI_U</code> functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	23
16	Computational times in seconds of the <code>LINSOLVETRI_U_VEC</code> function with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	23
17	Computational times in seconds of <code>CHOLESKY</code> functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	29
18	Computational times in seconds of the <code>CHOLESKY_VEC</code> function with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	29
19	Function <code>CHOLESKY_VEC</code> with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ under Matlab 2018a: effect of multithreaded on cputimes	29
20	Computational times in seconds of <code>PALU</code> functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	35

21	Computational times in seconds of <code>PALU_VEC</code> functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	35
22	Function <code>PALU_VEC</code> with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ under Matlab 2018a: effect of multithreaded on <code>cputimes</code>	35
23	Computational times in seconds of <code>PLUINPLACE</code> functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	38
24	Computational times in seconds of <code>PLUINPLACE_VEC</code> functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	38
25	Function <code>PLUINPLACE_VEC</code> with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ under Matlab 2018a: effect of multithreaded on <code>cputimes</code>	38
26	Computational times in seconds of the <code>LINSOLVECHOLESKY_VEC</code> function with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $B \in (\mathcal{M}_{3,1}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	40
27	Computational times in seconds of the <code>LINSOLVEPALU_VEC</code> function with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ and $B \in (\mathcal{M}_{3,1}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. The last column is for the native python function <code>numpy.linalg.solve</code>	42
28	Computational times in seconds of <code>DETLAPIDX</code> functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	46
29	Computational times in seconds of <code>DETLAP_VEC</code> functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. The last column is for the native python function <code>numpy.linalg.det</code>	46
30	Computational times in seconds of <code>DETLAPIDX_VEC</code> functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. The last column is for the native python function <code>numpy.linalg.det</code>	46
31	Computational times in seconds of <code>DETPLUIN_VEC</code> function with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. The last column is for the native python function <code>numpy.linalg.det</code>	48
32	Computational times in seconds of <code>DETMIX</code> functions with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.	51

References

[1] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2013.

[2] P. Lascaux and R. Théodor. *Analyse numérique matricielle appliquée à l'art de l'ingénieur*. Number vol. 1 in *Analyse numérique matricielle appliquée à l'art de l'ingénieur*. Dunod, 2004.

[3] Omid Rezaifar and Hossein Rezaee. A new approach for finding the determinant of matrices. *Applied Mathematics and Computation*, 188(2):1445–1454, 2007.