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
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.
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 Sub2ind function 53
    A.1.2 Ind2sub function 53
  A.2 Combinatorial functions 53

B Information for developers 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: [link]

Octave: [link]

Python: [link]

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 cputime 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:

- Z, N, R, C are respectively the set of integers, positive integers, reals and complex numbers. K is either R or C.

- All vectors or 1D-arrays are represented in bold: $v \in \mathbb{R}^n$ or $X$ a 1D-array. The first alphabetic characters are aAbBeC . . . .

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: $M \in \mathcal{M}_{m,n}(K)$ or $b$ a $m$-by-$n$ 2D-array. The first alphabetic characters are $a\text{AbBeC}$. .

All arrays of matrices or 3D-arrays are represented with the bold blackboard font as: $M \in (\mathcal{M}_{m,n}(K))^N$ or $b$ a $m$-by-$n$-by-$p$ 3D-array. The first alphabetic characters are $a\text{AbBeC}$. .

We now introduce some notations. Let $\mathbf{A} = (\mathbf{A}_1, \ldots, \mathbf{A}_N) \in (\mathcal{M}_{m,n}(K))^N$ where $N \gg m, n$. We identify $\mathbf{A}$ with a $N$-by-$m$-by-$n$ 3D-array such that

$$\mathbf{A}(k, :, :) = \mathbf{A}_k, \quad \forall k \in [1, N]. \quad (1)$$

Thereafter, we said that a 3D-array $\mathbf{A} \in (\mathcal{M}_{m,n}(K))^N$ has a matricial property if all the $\mathbf{A}_k$ 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, substraction, 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 (>= 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[4] some examples of operators are provided.

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

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

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

where $\forall k \in [1, N]$,

$$\forall i \in [1, m], \quad \forall j \in [1, n], \quad C_k(i,j) = A_k(i,j) \diamond B_k(i,j).$$
Table 1: Common element by element operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Algorithm</th>
<th>Matlab/Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiply</td>
<td>.*</td>
<td>times</td>
<td>* * multiply</td>
</tr>
<tr>
<td>divide</td>
<td>./</td>
<td>rdivide</td>
<td>./ / divide</td>
</tr>
<tr>
<td>add</td>
<td>+ +</td>
<td>plus</td>
<td>+ + add</td>
</tr>
<tr>
<td>subtract</td>
<td>- -</td>
<td>minus</td>
<td>- - subtract</td>
</tr>
<tr>
<td>power</td>
<td>.*</td>
<td>power</td>
<td>** ** power</td>
</tr>
<tr>
<td>divide</td>
<td>\ \</td>
<td>ldivide</td>
<td>\ \ ldivide</td>
</tr>
</tbody>
</table>

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

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

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

where $\forall k \in [1, N]$

$$\forall i \in [1, m], \forall j \in [1, n], \left\{ \begin{array}{l}
C_k(i, j) = \mathbf{A}_k(i, j) \odot \mathbb{B}(i, j), \\
D_k(i, j) = \mathbb{B}(i, j) \odot \mathbf{A}_k(i, j).
\end{array} \right.$$ (9)

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

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

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

where $\forall k \in [1, N]$

$$\forall i \in [1, m], \forall j \in [1, n], \left\{ \begin{array}{l}
C_k(i, j) = \mathbf{A}_k(i, j) \odot \mathbb{B}(k), \\
D_k(i, j) = \mathbb{B}(k) \odot \mathbf{A}_k(i, j).
\end{array} \right.$$ (10)

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

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

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

where $\forall k \in [1, N]$

$$\forall i \in [1, m], \forall j \in [1, n], \left\{ \begin{array}{l}
C_k(i, j) = \mathbf{A}_k(i, j) \odot \mathbb{B}, \\
D_k(i, j) = \mathbb{B} \odot \mathbf{A}_k(i, j).
\end{array} \right.$$ (11)

1. Linear Algebra

1.2 Determinants

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

$$D(k) = \det(k, \cdot, \cdot), \ \forall k \in [1, N].$$ (12)
1.2.2 Matricial products

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

\[
X \ast Y = Z \in (M_{m,p}(K))^N
\]  

(10)

where \( \forall k \in [1, N] \)

\[
Z(k, :,:) = \begin{cases} 
X(k, :, :) \ast Y(k, :, :), & \text{if } X \in (M_{m,n}(K))^N \text{ and } Y \in (M_{n,p}(K))^N, \\
X(k, :, :) \ast Y, & \text{if } X \in (M_{m,n}(K))^N \text{ and } Y \in M_{n,p}(K), \\
X \ast Y(k, :, :), & \text{if } X \in M_{m,n}(K) \text{ and } Y \in (M_{n,p}(K))^N. 
\end{cases}
\]

In these formulas, the operator \( \ast \) denotes the matricial product between a \( m \)-by-\( n \) matrix and a \( n \)-by-\( p \) matrix.

1.2.3 Linear systems

Let \( A \in (M_{m,n}(K))^N \) and let \( B \) be in \( (M_{n,p}(K))^N \) or in \( M_{n,p}(K) \). We want to find \( X \in (M_{m,p}(K))^N \) such that

\[
A \ast X = B
\]  

(11)

that is to say, for all \( k \in [1, N] \), find \( X_k \in M_{d,n}(K) \) solution of the linear system

\[
A_k \ast X_k = \begin{cases} 
B_k, & \text{if } B \in (M_{n,p}(K))^N, \\
B, & \text{if } B \in M_{n,p}(K). 
\end{cases}
\]

1.2.4 Positive Cholesky factorization

Let \( A \in (M_{d,d}(C))^N \) be a hermitian positive definite 3D-array:

\( \forall k \in [1, N] \), \( A(k, :, :) = A_k \in M_n(C) \) is a hermitian positive definite matrix.

The positive Cholesky factorization of \( A \) is given by

\[
A = LL^* 
\]  

(12)

where \( L \in (M_{d,d}(C))^N \) is lower triangular with real and positive diagonal entries (i.e. \( \forall k \in [1, N] \), \( L(k, :, :) = L_k \in M_n(C) \) are lower triangular matrices with real and positive diagonal entries) The equation [12] can be equivalently written as

\[
\forall k \in [1, N], A_k = L_k L_k^*. 
\]

1.2.5 LU factorization with partial pivoting

Let \( A \in (M_{d,d}(K))^N \). The LU factorization with partial pivoting of \( A \) is given by

\[
PA = LU
\]  

(13)

where \( L, U \) and \( P \) are in \( (M_{d,d}(K))^N \) and

- \( L \) is a lower triangular 3D-array with unit diagonal, i.e. \( \forall k \in [1, N] \), \( L(k, :, :) = L_k \in M_d(K) \) are lower triangular matrices with unit diagonal,
• $U$ is a upper triangular 3D-array, i.e. $\forall k \in [1, N], U(k, :, :) = U_k \in \mathcal{M}_d(K)$ are upper triangular matrices,

• $P$ is a permutation 3D-array, i.e. $\forall k \in [1, N], P(k, :, :) = P_k \in \mathcal{M}_d(K)$ are permutation matrices,

So we have, for all $k \in [1, N],$

$$P_k A_k = L_k U_k$$

\section{Basic Linear Algebra Vectorized operations}

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

• $AXpY\_Cpt$, component by component computation (3 loops)

• $AXpY\_Mat$, using 2D-array or matricial operations (1 loop over number of matrices),

• $AXpY\_Vec$, vectorized algorithm (2 loops over rows and columns of the matrices),

• $AXpY\_Cvt$, 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 $\mathbf{mtimes}$.

\subsection{Linear Combinations}

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

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

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

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

• with Matlab (\texttt{>=2016b}) and Octave, one has to expand $Y$ to a 1-by-$m$-by-$n$ 3D-array to fit broadcasting rules by using $\mathbf{reshape}$ function:

\begin{verbatim}
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;
\end{verbatim}
• with Matlab (<2016b), one has to expand $Y$ to a $N$-by-$m$-by-$n$ 3D-array by using `reshape` and `repmat` functions:

$$N = 10^5; m = 3; n = 2;$$
$$X = 	ext{randn}(N, m, n); Y = 	ext{randn}(m, n); Z = 	ext{randn}(N, 1);$$
$$C = 7 \times X - 5 \times 	ext{repmat} (\text{reshape}(Y, [1, m, n]), N, 1, 1);$$
$$D = 3 \times X + 4 \times \text{repmat}(Z, 1, m, n);$$

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

```python
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 \times X - 5 \times Y
D = 3 \times X + 4 \times 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 $Z$ is done component by component.

### Algorithm 1

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

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

Function $Z = \text{AXpBy}_Cpt(\alpha, X, \beta, Y)$
for $k = 1$ to $N$
  for $i = 1$ to $m$
    for $j = 1$ to $n$
      $Z(k, i, j) = \alpha \times \text{getCpt}(X, k, i, j) + \beta \times \text{getCpt}(Y, k, i, j)$
    end for
  end for
end for

end Function
```

### Algorithm 2

Function `getCpt`, returns component $(i, j)$ of the $k$th matrix of $X$.

```plaintext
Algorithm 2
Function `getCpt`, returns component $(i, j)$ of the $k$th matrix of $X$.

Input $X$: in $\mathbb{M}_{m,n}(K)$ or in $\mathbb{K}^n$ or in $K$.
$k$: matrix index,
$i$: row index,
$j$: column index
Output $s$: a scalar.

Function $s = \text{getCpt}(X, k, i, j)$
if $X \in K$ then
  $s = X$
else if $X \in \mathbb{K}^n$ then
  $s = X(k)$
else if $X \in \mathbb{M}_{m,n}(K)$ then
  $s = X(i, j)$
else
  $s = \text{vectorize}(X(k, i, j))$
end if

end Function
```

We present in Algorithm 2 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 \texttt{AXBY\_CVT}, returns linear combination $\alpha X + \beta Y$ by converting arrays to 3D-arrays.

Function $Z \leftarrow \texttt{AXBY\_CVT}(\alpha, X, \beta, Y)$

\begin{verbatim}
    Z \leftarrow \alpha \ast \texttt{TO3DARRAY}(X, N, m, n) + \beta \ast \texttt{TO3DARRAY}(Y, N, m, n)
\end{verbatim}

end Function

Algorithm 4 Function \texttt{TO3DARRAY}, converts to a 3D-array

Function $T \leftarrow \texttt{TO3DARRAY}(X, N, m, n)$

\begin{verbatim}
    if $X$ is in $K$ then
        $T \leftarrow X \ast \texttt{ONES}(N, m, n)$
    else if $X$ is in $K^N$ then
        $T \leftarrow \texttt{REPEAT}(X, 1, m, n)$
    else if $X$ is in $M_{m,n}(K)$ then
        $T \leftarrow \texttt{REPEAT}(X, N, 1, 1)$
    else
        $T \leftarrow X$
    end if
end Function

Algorithm 5 Function \texttt{AXBY\_MAT}, returns linear combination $\alpha X + \beta Y$ by using vectorized operations on 2D-arrays or matrices.

Function $Z \leftarrow \texttt{AXBY\_MAT}(\alpha, X, \beta, Y)$

\begin{verbatim}
    for $k \leftarrow 1$ to $N$
        $Z(k, :) \leftarrow \alpha \ast \texttt{GETMAT}(X, k) + \beta \ast \texttt{GETMAT}(Y, k)$
    end for
end Function

Algorithm 6 Function \texttt{GETMAT}, returns the $k$-th matrix of $X$.

Function $M \leftarrow \texttt{GETMAT}(X, k)$

\begin{verbatim}
    if $X$ is in $K$ or $X$ is in $M_{m,n}(K)$ then
        $M \leftarrow X(k)$
    else if $X$ is in $K^N$ then
        $M \leftarrow \texttt{X}(k)$
    else
        $M \leftarrow X(k, :)$
    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 \texttt{AXBY\_VEC}, returns linear combination $\alpha X + \beta Y$ by using vectorized operations on 3D-arrays.

Function $Z \leftarrow \texttt{AXBY\_VEC}(\alpha, X, \beta, Y)$

\begin{verbatim}
    for $i \leftarrow 1$ to $m$
        for $j \leftarrow 1$ to $n$
            $Z(i, j) \leftarrow \alpha \ast \texttt{GETVEC}(X, i, j) + \beta \ast \texttt{GETVEC}(Y, i, j)$
        end for
    end for
end Function

Algorithm 8 Function \texttt{GETVEC}, returns $(i, j)$ components of $X$.

Function $V \leftarrow \texttt{GETVEC}(X, i, j)$

\begin{verbatim}
    if $X$ is in $K^N$ or in $K$ then
        $V \leftarrow X(i, j)$
    else if $X$ is in $M_{m,n}(K)$ then
        $V \leftarrow \texttt{X}(i, j)$
    else
        $V \leftarrow X(i, j, :)$
    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.
In Table 2, the computation time in second of theses four functions under Matlab, Octave and Python are given when the input arrays are $X \in (\mathcal{M}_{3,3}(K))^N$ and $Y \in \mathcal{M}_{3,3}(K)$ and with a number $N$ up to $10^5$ for the slower function $\text{AXpy}_N$ and up to $10^6$ for the others. As expected the two functions $\text{AXpy}_V$ and $\text{AXpy}_C$ 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 $\text{AXpy}_N$. In Table 3 and 4, computation time in second of the $\text{AXpy}_N$ function and for $N$ up to $10^5$ are given when the input array $X \in (\mathcal{M}_{3,3}(K))^N$ and the input array $Y$ respectively in $\mathcal{M}_{3,3}(R)$ and in $(\mathcal{M}_{3,3}(K))^N$. In Tables 5 to 8, effects of multi-threading with Matlab are provided respectively for the functions $\text{AXpy}_N$, $\text{AXpy}_V$, $\text{AXpy}_C$ and $\text{AXpy}_N$.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 000</td>
<td>0.025(s)</td>
<td>4.829(s)</td>
<td>0.176(s)</td>
</tr>
<tr>
<td>40 000</td>
<td>0.039(s)</td>
<td>9.683(s)</td>
<td>0.351(s)</td>
</tr>
<tr>
<td>60 000</td>
<td>0.057(s)</td>
<td>14.520(s)</td>
<td>0.537(s)</td>
</tr>
<tr>
<td>80 000</td>
<td>0.078(s)</td>
<td>19.439(s)</td>
<td>0.717(s)</td>
</tr>
<tr>
<td>100 000</td>
<td>0.096(s)</td>
<td>24.237(s)</td>
<td>0.899(s)</td>
</tr>
</tbody>
</table>

(a) Function $\text{AXpy}_C$  

<table>
<thead>
<tr>
<th>$N$</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.322(s)</td>
<td>5.368(s)</td>
<td>0.361(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.760(s)</td>
<td>11.349(s)</td>
<td>0.723(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>1.138(s)</td>
<td>17.118(s)</td>
<td>1.094(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>1.544(s)</td>
<td>22.823(s)</td>
<td>1.459(s)</td>
</tr>
<tr>
<td>1000 000</td>
<td>1.894(s)</td>
<td>28.534(s)</td>
<td>1.812(s)</td>
</tr>
</tbody>
</table>

(b) Function $\text{AXpy}_C$  

<table>
<thead>
<tr>
<th>$N$</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.008(s)</td>
<td>0.009(s)</td>
<td>0.015(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.011(s)</td>
<td>0.014(s)</td>
<td>0.033(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.020(s)</td>
<td>0.029(s)</td>
<td>0.059(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.032(s)</td>
<td>0.039(s)</td>
<td>0.081(s)</td>
</tr>
<tr>
<td>1000 000</td>
<td>0.040(s)</td>
<td>0.048(s)</td>
<td>0.102(s)</td>
</tr>
</tbody>
</table>

(c) Function $\text{AXpy}_M$  

<table>
<thead>
<tr>
<th>$N$</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.006(s)</td>
<td>0.003(s)</td>
<td>0.004(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.006(s)</td>
<td>0.009(s)</td>
<td>0.008(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.011(s)</td>
<td>0.034(s)</td>
<td>0.033(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.019(s)</td>
<td>0.045(s)</td>
<td>0.044(s)</td>
</tr>
<tr>
<td>1000 000</td>
<td>0.022(s)</td>
<td>0.055(s)</td>
<td>0.054(s)</td>
</tr>
</tbody>
</table>

(d) Function $\text{AXpy}_V$  

<table>
<thead>
<tr>
<th>$N$</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.008(s)</td>
<td>0.006(s)</td>
<td>0.005(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.011(s)</td>
<td>0.013(s)</td>
<td>0.012(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.012(s)</td>
<td>0.014(s)</td>
<td>0.013(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.013(s)</td>
<td>0.015(s)</td>
<td>0.014(s)</td>
</tr>
<tr>
<td>1000 000</td>
<td>0.017(s)</td>
<td>0.019(s)</td>
<td>0.018(s)</td>
</tr>
</tbody>
</table>

(e) Function $\text{AXpy}_N$  

<table>
<thead>
<tr>
<th>$N$</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.014(s)</td>
<td>0.014(s)</td>
<td>0.013(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.017(s)</td>
<td>0.026(s)</td>
<td>0.018(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.030(s)</td>
<td>0.089(s)</td>
<td>0.049(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.038(s)</td>
<td>0.118(s)</td>
<td>0.065(s)</td>
</tr>
<tr>
<td>1000 000</td>
<td>0.045(s)</td>
<td>0.146(s)</td>
<td>0.082(s)</td>
</tr>
</tbody>
</table>

Table 2: Computational times in seconds of $\text{AXpy}$ functions with $X \in (\mathcal{M}_{3,3}(K))^N$ and $Y \in \mathcal{M}_{3,3}(K)$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.
<table>
<thead>
<tr>
<th>$N$</th>
<th>Matlab</th>
<th>Matlab(*)</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.006(s)</td>
<td>0.009(s)</td>
<td>0.006(s)</td>
<td>0.004(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.007(s)</td>
<td>0.012(s)</td>
<td>0.011(s)</td>
<td>0.009(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.016(s)</td>
<td>0.035(s)</td>
<td>0.024(s)</td>
<td>0.022(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.019(s)</td>
<td>0.045(s)</td>
<td>0.045(s)</td>
<td>0.044(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.022(s)</td>
<td>0.056(s)</td>
<td>0.055(s)</td>
<td>0.055(s)</td>
</tr>
<tr>
<td>5 000 000</td>
<td>0.065(s)</td>
<td>0.268(s)</td>
<td>0.268(s)</td>
<td>0.273(s)</td>
</tr>
<tr>
<td>10 000 000</td>
<td>0.120(s)</td>
<td>0.521(s)</td>
<td>0.526(s)</td>
<td>0.545(s)</td>
</tr>
</tbody>
</table>

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

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

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

<table>
<thead>
<tr>
<th>$N$</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>6 threads</th>
<th>8 threads</th>
<th>14 threads</th>
<th>20 threads</th>
<th>28 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.382(s)</td>
<td>0.393(s)</td>
<td>0.396(s)</td>
<td>0.382(s)</td>
<td>0.376(s)</td>
<td>0.376(s)</td>
<td>0.376(s)</td>
<td>0.377(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.763(s)</td>
<td>0.762(s)</td>
<td>0.763(s)</td>
<td>0.755(s)</td>
<td>0.751(s)</td>
<td>0.750(s)</td>
<td>0.750(s)</td>
<td>0.762(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>1.136(s)</td>
<td>1.134(s)</td>
<td>1.141(s)</td>
<td>1.146(s)</td>
<td>1.168(s)</td>
<td>1.143(s)</td>
<td>1.134(s)</td>
<td>1.143(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>1.517(s)</td>
<td>1.525(s)</td>
<td>1.514(s)</td>
<td>1.511(s)</td>
<td>1.510(s)</td>
<td>1.502(s)</td>
<td>1.511(s)</td>
<td>1.518(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>1.921(s)</td>
<td>1.897(s)</td>
<td>1.897(s)</td>
<td>1.886(s)</td>
<td>1.890(s)</td>
<td>1.901(s)</td>
<td>1.899(s)</td>
<td>1.910(s)</td>
</tr>
<tr>
<td>5 000 000</td>
<td>9.582(s)</td>
<td>9.494(s)</td>
<td>9.458(s)</td>
<td>9.545(s)</td>
<td>9.516(s)</td>
<td>9.566(s)</td>
<td>9.514(s)</td>
<td>9.533(s)</td>
</tr>
<tr>
<td>10 000 000</td>
<td>19.406(s)</td>
<td>19.468(s)</td>
<td>19.383(s)</td>
<td>19.638(s)</td>
<td>19.480(s)</td>
<td>19.589(s)</td>
<td>19.705(s)</td>
<td>19.991(s)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$N$</th>
<th>1 thread</th>
<th>2 threads</th>
<th>4 threads</th>
<th>6 threads</th>
<th>8 threads</th>
<th>14 threads</th>
<th>20 threads</th>
<th>28 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.008(s)</td>
<td>0.005(s)</td>
<td>0.006(s)</td>
<td>0.006(s)</td>
<td>0.007(s)</td>
<td>0.007(s)</td>
<td>0.007(s)</td>
<td>0.006(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.014(s)</td>
<td>0.011(s)</td>
<td>0.011(s)</td>
<td>0.011(s)</td>
<td>0.012(s)</td>
<td>0.013(s)</td>
<td>0.013(s)</td>
<td>0.012(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.024(s)</td>
<td>0.024(s)</td>
<td>0.025(s)</td>
<td>0.025(s)</td>
<td>0.026(s)</td>
<td>0.026(s)</td>
<td>0.026(s)</td>
<td>0.026(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.032(s)</td>
<td>0.032(s)</td>
<td>0.032(s)</td>
<td>0.033(s)</td>
<td>0.033(s)</td>
<td>0.033(s)</td>
<td>0.033(s)</td>
<td>0.033(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.040(s)</td>
<td>0.039(s)</td>
<td>0.040(s)</td>
<td>0.039(s)</td>
<td>0.040(s)</td>
<td>0.040(s)</td>
<td>0.040(s)</td>
<td>0.040(s)</td>
</tr>
<tr>
<td>5 000 000</td>
<td>0.307(s)</td>
<td>0.292(s)</td>
<td>0.281(s)</td>
<td>0.277(s)</td>
<td>0.276(s)</td>
<td>0.276(s)</td>
<td>0.274(s)</td>
<td>0.274(s)</td>
</tr>
<tr>
<td>10 000 000</td>
<td>0.617(s)</td>
<td>0.584(s)</td>
<td>0.561(s)</td>
<td>0.554(s)</td>
<td>0.552(s)</td>
<td>0.557(s)</td>
<td>0.549(s)</td>
<td>0.531(s)</td>
</tr>
</tbody>
</table>

Table 6: Function $AXpY_{\text{Vec}}$ with $X \in (\mathcal{M}_{3,3}(K))^N$ and $Y \in \mathcal{M}_{3,3}(R)$ under Matlab 2018a: effects of multithreading on cputimes
From the four Algorithms 1, 3, 5, 7 and by using notations and definitions of section 2.1, we deduce four generic functions which computes $X \forall Y$. There are given in Algorithms 9 to 11.

<table>
<thead>
<tr>
<th>$X$</th>
<th>1 threads</th>
<th>2 threads</th>
<th>4 threads</th>
<th>6 threads</th>
<th>8 threads</th>
<th>14 threads</th>
<th>20 threads</th>
<th>28 threads</th>
<th>$Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.018</td>
<td>0.029</td>
<td>0.012</td>
<td>0.012</td>
<td>0.012</td>
<td>0.012</td>
<td>0.012</td>
<td>0.012</td>
<td>0.012</td>
</tr>
<tr>
<td>400 000</td>
<td>0.030</td>
<td>0.020</td>
<td>0.017</td>
<td>0.016</td>
<td>0.017</td>
<td>0.017</td>
<td>0.017</td>
<td>0.017</td>
<td>0.017</td>
</tr>
<tr>
<td>600 000</td>
<td>0.030</td>
<td>0.020</td>
<td>0.017</td>
<td>0.016</td>
<td>0.017</td>
<td>0.017</td>
<td>0.017</td>
<td>0.017</td>
<td>0.017</td>
</tr>
<tr>
<td>800 000</td>
<td>0.027</td>
<td>0.024</td>
<td>0.024</td>
<td>0.023</td>
<td>0.023</td>
<td>0.023</td>
<td>0.023</td>
<td>0.023</td>
<td>0.023</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.096</td>
<td>0.063</td>
<td>0.047</td>
<td>0.045</td>
<td>0.045</td>
<td>0.045</td>
<td>0.045</td>
<td>0.045</td>
<td>0.045</td>
</tr>
<tr>
<td>5 000 000</td>
<td>0.474</td>
<td>0.299</td>
<td>0.177</td>
<td>0.161</td>
<td>0.153</td>
<td>0.140</td>
<td>0.136</td>
<td>0.129</td>
<td>0.129</td>
</tr>
<tr>
<td>10 000 000</td>
<td>0.935</td>
<td>0.959</td>
<td>0.334</td>
<td>0.299</td>
<td>0.278</td>
<td>0.273</td>
<td>0.263</td>
<td>0.263</td>
<td>0.263</td>
</tr>
</tbody>
</table>

Table 7: Function $\text{AXbyY}_Cvt$ with $X \in (\mathcal{M}_{3,3}(K))^N$ and $Y \in \mathcal{M}_{3,3}(R)$ under Matlab 2018a; effects of multithreading on cpetimes

<table>
<thead>
<tr>
<th>$X$</th>
<th>1 threads</th>
<th>2 threads</th>
<th>4 threads</th>
<th>6 threads</th>
<th>8 threads</th>
<th>14 threads</th>
<th>20 threads</th>
<th>28 threads</th>
<th>$Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.016</td>
<td>0.006</td>
<td>0.005</td>
<td>0.005</td>
<td>0.005</td>
<td>0.005</td>
<td>0.005</td>
<td>0.005</td>
<td>0.005</td>
</tr>
<tr>
<td>400 000</td>
<td>0.015</td>
<td>0.008</td>
<td>0.007</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
</tr>
<tr>
<td>600 000</td>
<td>0.037</td>
<td>0.023</td>
<td>0.019</td>
<td>0.018</td>
<td>0.017</td>
<td>0.016</td>
<td>0.016</td>
<td>0.016</td>
<td>0.016</td>
</tr>
<tr>
<td>800 000</td>
<td>0.047</td>
<td>0.030</td>
<td>0.022</td>
<td>0.021</td>
<td>0.020</td>
<td>0.021</td>
<td>0.021</td>
<td>0.021</td>
<td>0.021</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.057</td>
<td>0.035</td>
<td>0.025</td>
<td>0.024</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>5 000 000</td>
<td>0.273</td>
<td>0.163</td>
<td>0.102</td>
<td>0.082</td>
<td>0.075</td>
<td>0.065</td>
<td>0.063</td>
<td>0.061</td>
<td>0.061</td>
</tr>
<tr>
<td>10 000 000</td>
<td>0.534</td>
<td>0.310</td>
<td>0.193</td>
<td>0.153</td>
<td>0.134</td>
<td>0.122</td>
<td>0.117</td>
<td>0.110</td>
<td>0.110</td>
</tr>
</tbody>
</table>

Table 8: Function $\text{AXbyY}_Nat$ with $X \in (\mathcal{M}_{3,3}(K))^N$ and $Y \in \mathcal{M}_{3,3}(R)$ under Matlab 2018a; effects of multithreading on cpetimes

2.2 Element by element operations

Thereafter, writing functions for a specific element by element operator is easy. For example, the corresponding functions for element by element multiplication
operator \( \cdot \) are provided in Algorithms 13 to 15.

### Algorithm 13 Function `times_Cvt`, returns element by element product \( X \cdot Y \)

Function \( Z \gets \text{times}_Cvt(X, Y) \)

\[ f : (x, y) \rightarrow x \cdot y \]

\( Z \gets \text{envr}_Cvt(X, Y; f) \)

end Function

### Algorithm 14 Function `times_Mat`, returns element by element product \( X \cdot Y \) by using vectorized operations on 2D-arrays or matrices.

Function \( Z \gets \text{times}_M a t(X, Y) \)

\[ f : (A, B) \rightarrow A \cdot B \]

\( Z \gets \text{envr}_M a t(X, Y; f) \)

end Function

### Algorithm 15 Function `times_Vec`, returns element by element product \( X \cdot Y \) by using vectorized operations on 3D-arrays.

Function \( Z \gets \text{times}_V e c(X, Y) \)

\[ f : (A, B) \rightarrow A \cdot B \]

\( Z \gets \text{envr}_V e c(X, Y; f) \)

end Function

### Algorithm 16 Function `times_Nat`, returns element by element product \( X \cdot Y \) by converting arrays to 3D-arrays.

Function \( Z \gets \text{times}_N a t(X, Y) \)

\[ f : (A, B) \rightarrow A \cdot B \]

\( Z \gets \text{envr}_N a t(X, Y; f) \)

end Function

In Table 9, the computation time in seconds of these four functions under Matlab, Octave, and Python are given when the input arrays are \( X \in (\mathcal{M}_{3,3}(K))^N \) and \( Y \in \mathcal{M}_{3,3}(R) \). For the `times_Cvt` 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`.

<table>
<thead>
<tr>
<th>( N )</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.009(s)</td>
<td>0.015(s)</td>
<td>0.004(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.017(s)</td>
<td>0.026(s)</td>
<td>0.012(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.031(s)</td>
<td>0.057(s)</td>
<td>0.030(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.038(s)</td>
<td>0.076(s)</td>
<td>0.040(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.044(s)</td>
<td>0.093(s)</td>
<td>0.051(s)</td>
</tr>
</tbody>
</table>

(a) Function `times_Cvt`

<table>
<thead>
<tr>
<th>( N )</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.007(s)</td>
<td>0.008(s)</td>
<td>0.015(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.011(s)</td>
<td>0.012(s)</td>
<td>0.034(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.023(s)</td>
<td>0.026(s)</td>
<td>0.055(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.030(s)</td>
<td>0.054(s)</td>
<td>0.080(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.033(s)</td>
<td>0.041(s)</td>
<td>0.101(s)</td>
</tr>
</tbody>
</table>

(b) Function `times_Mat`

<table>
<thead>
<tr>
<th>( N )</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.004(s)</td>
<td>0.002(s)</td>
<td></td>
</tr>
<tr>
<td>400 000</td>
<td>0.005(s)</td>
<td>0.005(s)</td>
<td></td>
</tr>
<tr>
<td>600 000</td>
<td>0.009(s)</td>
<td>0.018(s)</td>
<td>0.012(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.011(s)</td>
<td>0.023(s)</td>
<td>0.024(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.013(s)</td>
<td>0.029(s)</td>
<td>0.030(s)</td>
</tr>
</tbody>
</table>

(c) Function `times_Vec`

<table>
<thead>
<tr>
<th>( N )</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.009(s)</td>
<td>0.015(s)</td>
<td></td>
</tr>
<tr>
<td>400 000</td>
<td>0.012(s)</td>
<td>0.034(s)</td>
<td></td>
</tr>
<tr>
<td>600 000</td>
<td>0.023(s)</td>
<td>0.055(s)</td>
<td></td>
</tr>
<tr>
<td>800 000</td>
<td>0.030(s)</td>
<td>0.080(s)</td>
<td></td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.033(s)</td>
<td>0.101(s)</td>
<td></td>
</tr>
</tbody>
</table>

(d) Function `times_Nat`

Table 9: Computational times in seconds of `times` functions with \( X \in (\mathcal{M}_{3,3}(K))^N \) and \( Y \in \mathcal{M}_{3,3}(R) \) 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 $X \ast Y$ where $X$ or/and $Y$ are 3D-arrays.

Function $C \leftarrow \text{mtimes}_Cpt (X,Y)$
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
        $S \leftarrow S + \text{getCpt}_{p}(X,k,i,l) \ast \text{getCpt}_{p}(Y,k,l,j)$
      end for
      $C(i,j,k) \leftarrow S$
    end for
  end for
end for

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

Function $C \leftarrow \text{mtimes}_\text{Vec} (X,Y)$
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
      $S \leftarrow S + \text{getVec}_{p}(X,i,l) \ast \text{getVec}_{p}(Y,l,j)$
    end for
    $C(i,j) \leftarrow S$
  end for
end for

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

Function $C \leftarrow \text{mtimes}_\text{Mat} (X,Y)$
for $k \leftarrow 1$ to $N$ do
  $C(k,:) \leftarrow \text{getMat}_{p}(X,k) \ast \text{getMat}_{p}(Y,k)$
end for

In Python, the `matmul` function of the Numpy package (>= 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 these three functions under Matlab, Octave and Python are given when the input arrays are $X \in (M_{3,3}(K))^N$ and $Y \in M_{3,3}(R)$. 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 computational 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 computational time of the same functions are given but for $X$ and $Y$ in $(M_{3,3}(K))^N$. 

14
Table 10: Computational times in seconds of \texttt{mtimes} functions with $X \in (\mathcal{M}_{3,3}(K))^N$ and $Y \in \mathcal{M}_{3,3}(R)$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Matlab</th>
<th>Matlab(*)</th>
<th>Octave</th>
<th>Python</th>
<th>Python(Nat)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.587(s)</td>
<td>0.355(s)</td>
<td>1.456(s)</td>
<td>0.164(s)</td>
<td>0.325(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>1.198(s)</td>
<td>0.713(s)</td>
<td>3.258(s)</td>
<td>2.034(s)</td>
<td>0.602(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>1.809(s)</td>
<td>1.070(s)</td>
<td>4.856(s)</td>
<td>3.453(s)</td>
<td>0.987(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>2.415(s)</td>
<td>1.458(s)</td>
<td>6.459(s)</td>
<td>5.278(s)</td>
<td>1.298(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>3.015(s)</td>
<td>1.859(s)</td>
<td>8.158(s)</td>
<td>6.678(s)</td>
<td>1.598(s)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$N$</th>
<th>Matlab</th>
<th>Matlab(*)</th>
<th>Octave</th>
<th>Python</th>
<th>Python(Nat)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.471(s)</td>
<td>0.251(s)</td>
<td>0.331(s)</td>
<td>0.009(s)</td>
<td>0.008(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.951(s)</td>
<td>0.456(s)</td>
<td>0.453(s)</td>
<td>0.012(s)</td>
<td>0.008(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>1.395(s)</td>
<td>0.711(s)</td>
<td>0.657(s)</td>
<td>0.034(s)</td>
<td>0.015(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>1.899(s)</td>
<td>1.073(s)</td>
<td>0.874(s)</td>
<td>0.030(s)</td>
<td>0.008(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>2.325(s)</td>
<td>2.093(s)</td>
<td>1.092(s)</td>
<td>0.101(s)</td>
<td>0.006(s)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$N$</th>
<th>Matlab</th>
<th>Matlab(*)</th>
<th>Octave</th>
<th>Python</th>
<th>Python(Nat)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.346(s)</td>
<td>0.178(s)</td>
<td>0.227(s)</td>
<td>0.051(s)</td>
<td>0.078(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.088(s)</td>
<td>0.078(s)</td>
<td>0.149(s)</td>
<td>0.117(s)</td>
<td>0.159(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.098(s)</td>
<td>0.099(s)</td>
<td>0.167(s)</td>
<td>0.182(s)</td>
<td>0.242(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.133(s)</td>
<td>0.139(s)</td>
<td>0.215(s)</td>
<td>0.256(s)</td>
<td>0.327(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.264(s)</td>
<td>0.255(s)</td>
<td>0.298(s)</td>
<td>0.322(s)</td>
<td>0.405(s)</td>
</tr>
<tr>
<td>5 000 000</td>
<td>1.167(s)</td>
<td>1.678(s)</td>
<td>1.079(s)</td>
<td>1.689(s)</td>
<td>2.032(s)</td>
</tr>
<tr>
<td>10 000 000</td>
<td>2.961(s)</td>
<td>3.290(s)</td>
<td>3.073(s)</td>
<td>3.697(s)</td>
<td>4.065(s)</td>
</tr>
</tbody>
</table>
Figure 1: Computational times in seconds of the \texttt{mtimes\_Vec} function with $X$ and $Y$ both in $(\mathbb{M}_{d,d}(\mathbb{K}))^N$, $N = 10^6$ and $d \in [2, 10]$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

Figure 2: Computational times in seconds of the \texttt{mtimes\_Vec} function with $X \in (\mathbb{M}_{d,d}(\mathbb{K}))^N$, $Y \in \mathbb{M}_{d,d}(\mathbb{R})$, $N = 10^6$ and $d \in [2, 10]$ 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 $\mathbf{A}_k$ 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

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

In this section we give the $\texttt{LinsolveDiag}$ function and various versions of the $\texttt{LinsolveTriL}$ and $\texttt{LinsolveTriU}$ 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 $\mathbf{D} \in \mathcal{M}_{d,d}(\mathbb{K})$ 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 $\mathbf{D} \mathbf{x} = \mathbf{b}$ is given by

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

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

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

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 $\mathbf{A}_k \equiv \mathbf{A}(k, :, :)$ is a regular diagonal matrix, and so $\forall k \in [1,N], \forall (i,j) \in [1,d]^2$

$$\mathbf{A}_k(i,j) = 0, \text{ if } i \neq j$$
$$\mathbf{A}_k(i,i) \neq 0.$$

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

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

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

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

By using the function $\texttt{getVec}$ described in Algorithm 8 we obtain the vectorized Algorithm 20.
Algorithm 20 Function LINSOLVE_DIAG, solves diagonal linear system \( AX = B \).

**Input**
- \( A \) : in \((\mathcal{M}_{d,d}(K))^N\)
- \( B \) : in \((\mathcal{M}_{d,n}(K))^N\), or in \(\mathcal{M}_{d,n}(K)\)

**Output**
- \( X \) : in \((\mathcal{M}_{d,n}(K))^N\)

```plaintext
Function \( X \leftarrow LINSOLVE_DIAG (A, B) \)
for \( l \leftarrow 1 \) to \( m \) do
  for \( i \leftarrow 1 \) to \( d \) do
    \( X(:,i,l) \leftarrow \text{GETVEC}(B,i,l)/A(:,i,i) \)
  end for
end for
```

### 3.2 Lower triangular matrices

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

\[
AX = B
\]

can be computed column by column. For each column \( l \), we successively compute \( X_{1,l}, X_{2,l}, \ldots, X_{d,l} \) by using formula

\[
X_{i,l} = (E_{i,l} - \sum_{j=1}^{i-1} A_{i,j} X_{j,l})/A_{i,i}, \quad \forall i \in [1,d], \forall l \in [1,n].
\]

or in a more compact form we successively compute \( X_{1,:}, X_{2,:), \ldots, X_{d,:} \) by using formula

\[
X_{i,:} = (E_{i,:} - A_{i,1:d-1} X_{1:d-1,:})/A_{i,i}, \quad \forall i \in [1,d].
\]

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

Now, one can extend these results to **regular lower triangular 3D-array**. Let \( A \in (\mathcal{M}_{d,d}(K))^N \) be a regular lower triangular 3D-array, i.e. each \( A_{k} \overset{\text{def}}{=} A(k,\cdot,\cdot) \) is a regular lower triangular matrix, and so \( \forall k \in [1,N], \forall (i,j) \in [1,d]^2 \)

\[
A_k(i,j) = 0, \text{ if } i < j \quad A_k(i,i) \neq 0.
\]

By using LINSOLVE_TRI and GETMAT functions respectively described in Algorithm 21 and Algorithm 6 we easily obtain the non-vectorized function LINSOLVE_TRI MAT written in Algorithm 22. In Algorithm 23 an other code is presented without using function LINSOLVE_TRI 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 LINSOLVE_TRI_VEC given in Algorithm 24.
computation time in second of the Linsol veTriL_Vec

As expected the function Octave and Python are given with veTriL_Cpt or in a more compact form we successively compute triangular matrix. If \( B \) is a regular lower triangular matrix

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

\[
\mathbf{AX} = \mathbf{B}
\]

can be computed column by column. For each column \( l \), we successively compute \( X_{d,l}, X_{d-1,l}, \ldots, X_{1,l} \) by using formula

\[
X_{i,l} = (B_{i,l} - \sum_{j=i+1}^{d} A_{i,j} X_{j,l})/A_{i,i}, \quad \forall i \in [1, d], \forall l \in [1, n].
\]

or in a more compact form we successively compute \( X_{d,:}, X_{d-1,:), \ldots, X_{1,:} \) by using formula

3.3 Upper triangular matrices

Finally, we give in Figure 3 computational times in second of the Linsol veTriL_Vec function is given for \( N \) values up to \( 10^6 \). Finally, we give in Figure 3 computational times in second of the Linsol veTriL_Vec with \( \mathbf{A} \in (\mathcal{M}_{d,d}(\mathbf{K}))^N \), \( \mathbf{B} \in (\mathcal{M}_{d,1}(\mathbf{K}))^N \), for \( N = 10^6 \) and \( d \in [2, 10] \).
### Table 13: Computational times in seconds of \texttt{LinsolveTriL} functions with \( A \in (M_{3,3}(K))^N \) for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

<table>
<thead>
<tr>
<th>( N )</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.798(s)</td>
<td>17.996(s)</td>
<td>1.566(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>1.632(s)</td>
<td>36.272(s)</td>
<td>3.131(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>2.481(s)</td>
<td>54.323(s)</td>
<td>4.687(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>3.286(s)</td>
<td>72.349(s)</td>
<td>6.237(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>4.016(s)</td>
<td>90.541(s)</td>
<td>7.814(s)</td>
</tr>
</tbody>
</table>

### Table 14: Computational times in seconds of the \texttt{LinsolveTriL}_{\text{vec}} function with \( A \in (M_{3,3}(K))^N \) for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

<table>
<thead>
<tr>
<th>( N )</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.011(s)</td>
<td>0.007(s)</td>
<td>0.008(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.010(s)</td>
<td>0.012(s)</td>
<td>0.019(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.015(s)</td>
<td>0.021(s)</td>
<td>0.030(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.018(s)</td>
<td>0.021(s)</td>
<td>0.041(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.023(s)</td>
<td>0.027(s)</td>
<td>0.052(s)</td>
</tr>
<tr>
<td>5 000 000</td>
<td>0.255(s)</td>
<td>0.247(s)</td>
<td>0.381(s)</td>
</tr>
<tr>
<td>10 000 000</td>
<td>0.503(s)</td>
<td>0.612(s)</td>
<td>0.763(s)</td>
</tr>
</tbody>
</table>
Figure 3: Computational times in seconds of the \texttt{Linsolve\_tri\_Vec} function with $A \in (\mathcal{M}_{d,d}(K))^N$, $B \in (\mathcal{M}_{d,1}(K))^N$, $N = 10^6$ and $d \in [2, 10]$ 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 \( A \in (M_d,d(K))^N \) be a regular upper triangular 3D-array, i.e. each \( A_{k}^{d} = A(k,:,:) \) is a regular upper triangular, and so \( \forall k \in [1,N], \forall (i,j) \in [1,d]^2 \):

\[
\begin{align*}
A_k(i,j) &= 0, \text{ if } i > j, \\
A_k(i,i) &\neq 0.
\end{align*}
\]

By using LinsolveTriU and getMat functions respectively described in Algorithm 25 and Algorithm 26 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.

\begin{verbatim}
Algorithm 26: Function LinsolveTriU_Mat, solves equation \( AX = B \) where \( A \) is a regular upper triangular 3D-array (not vectorized).
Input \( A \) : in \( (M_d,d(K))^N \), \( B \) : in \( M_d(d(K))^N \), or in \( M_d,0(K) \).
Output \( X \) : in \( (M_d,d(K))^N \).

\begin{verbatim}
function X = LinsolveTriU_Mat(A, B)
    for k = 1 to N do
        for i = 1 to n do
            S = 0
            for j = i - d to 1 (step -1) do
                S = S + A(k,j,i) * (B(k,j,i) + B(k,j))
            end for
            X(k,i,i) = (B(k,i,i) - S) / A(k,i,i)
        end for
    end for
end function
\end{verbatim}
\end{verbatim}

\begin{verbatim}
Algorithm 27: Function LinsolveTriU_Cpt, solves equation \( AX = B \) where \( A \) is a regular upper triangular 3D-array (not vectorized).

function X = LinsolveTriU_Cpt(A, B)
    for k = 1 to N do
        for i = 1 to n do
            for j = i - d to 1 (step -1) do
                S = S + A(k,j,i) * (B(k,j,i) + B(k,j))
            end for
            X(k,i,i) = (B(k,i,i) - S) / A(k,i,i)
        end for
    end for
end function
\end{verbatim}

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 \( A \in (M_{3,3}(K))^N \) and \( B \in (M_{3,1}(K))^N \). 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 10^6. Finally, we give in Figure 4 computational times in second of the LinsolveTriU_Vec with \( A \in (M_{5,5}(K))^N \), \( B \in (M_{5,1}(K))^N \), for \( N = 10^6 \) and \( d \in [2, 10] \).
Table 15: Computational times in seconds of \texttt{LinsolveTriU} functions with $A \in (\mathcal{M}_{3,3}(K))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Matlab 2018a</th>
<th>Octave 4.4.0</th>
<th>Python 3.6.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.007(s)</td>
<td>0.006(s)</td>
<td>0.008(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.010(s)</td>
<td>0.010(s)</td>
<td>0.021(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.021(s)</td>
<td>0.018(s)</td>
<td>0.030(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.018(s)</td>
<td>0.029(s)</td>
<td>0.041(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.023(s)</td>
<td>0.026(s)</td>
<td>0.052(s)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$N$</th>
<th>Matlab 2018a</th>
<th>Octave 4.4.0</th>
<th>Python 3.6.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.011(s)</td>
<td>0.006(s)</td>
<td>0.009(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.011(s)</td>
<td>0.011(s)</td>
<td>0.020(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.015(s)</td>
<td>0.018(s)</td>
<td>0.031(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.019(s)</td>
<td>0.020(s)</td>
<td>0.042(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.023(s)</td>
<td>0.026(s)</td>
<td>0.053(s)</td>
</tr>
<tr>
<td>5 000 000</td>
<td>0.255(s)</td>
<td>0.239(s)</td>
<td>0.388(s)</td>
</tr>
<tr>
<td>10 000 000</td>
<td>0.503(s)</td>
<td>0.603(s)</td>
<td>0.771(s)</td>
</tr>
</tbody>
</table>
Figure 4: Computational times in seconds of the `linsolve_triu_vec` function with $A \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$, $B \in (\mathcal{M}_{d,1}(\mathbb{K}))^N$, $N = 10^6$ and $d \in [2, 10]$ 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 $B$ be a hermitian, positive-definite matrix in $M_n(C)$. One can apply the Cholesky factorization: there exists a unique lower triangular matrix $L \in M_n(C)$ with real and positive diagonal entries such that

$$B = LL^*.$$  \hspace{1cm} (21)

With this factorization, the determinant of the matrix $B$ computes easily as $L$ is a lower triangular matrix:

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

To compute the lower triangular matrix $L$ we use the following formula

$$L_{j,j} = \sqrt{B_{j,j} - \sum_{k=1}^{j-1} L_{j,k}L_{j,k}^*}$$

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

In Algorithm 29 the Cholesky-Crout algorithm is given to compute the matrix $L$. It starts from the upper left corner of the matrix $L$ and proceeds to calculate the matrix column by column.
Algorithm 29 Function Cholesky. Computes the lower triangular matrix \( L \in \mathcal{M}_n(C) \) such that \( B = LL^* \).

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

Résultat: \( L \): the lower triangular matrix \( L \in \mathcal{M}_n(C) \) with \( L(i, i) > 0, \forall i \in [1, n] \).

1: Function \( L \leftarrow \text{Cholesky} \left( B \right) \)
2: \( L \leftarrow \text{Zeros}(n, n) \)
3: for \( j \leftarrow 1 \) to \( n \) do \( \quad \Rightarrow \) Computes the \( j \)-th column of \( L \)
4: \( S_1 \leftarrow 0 \)
5: for \( l \leftarrow 1 \) to \( j - 1 \) do
6: \( S_1 \leftarrow S_1 + |L(j, l)|^2 \)
7: end for
8: \( L(j, j) \leftarrow \text{sqr}(B(j, j) - S_1) \)
9: for \( i \leftarrow j + 1 \) to \( n \) do
10: \( S_2 \leftarrow 0 \)
11: for \( l \leftarrow 1 \) to \( j - 1 \) do
12: \( S_2 \leftarrow S_2 + L(i, l) \ast \text{Conj}(L(j, l)) \)
13: end for
14: \( L(i, j) \leftarrow (B(i, j) - S_2)/L(j, j) \).
15: end for
16: end for
17: end Function

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

\[
\forall k \in [1, N], \quad A(k, ::, :) = A_k \in \mathcal{M}_n(C) \text{ is a hermitian positive definite matrix}
\]

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

\[
\forall k \in [1, N], A_k = L_k L_k^*
\]

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 \( N \geq d \), vectorization of the Algorithm 31 consists to firstly permute the \( k \) loop over \([1, N]\) with the \( j \) loop over \([1, d]\) 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.

26
Algorithm 30 Function \texttt{Cholesky\_Mat}, returns cholesky factorizations of $A_k$ matrices (not vectorized)

\textbf{Input} $A$ : $N$-by-$d$-by-$d$ 3D array such that $A(k,:,:)=A_k, \forall k \in [1,N]$.

\textbf{Output} $L$ : $N$-by-$d$-by-$d$ 3D array such that $\forall k \in [1,N], L_k \equiv L(k,:,:)$ and $A_k = L_k \cdot L_k^\ast$.

\begin{algorithm}
\begin{algorithmic}
\Function{$L \leftarrow \texttt{Cholesky\_Mat}(A)$}
\For{$k \leftarrow 1$ \textbf{to} $N$}
\State $L(k,:,:) \leftarrow \texttt{Cholesky}(A(k,:,:))$
\EndFor
\EndFunction
\end{algorithmic}
\end{algorithm}

Algorithm 31 Function \texttt{Cholesky\_Cpt}, returns cholesky factorizations of $A_k$ matrices (not vectorized)

\begin{algorithm}
\begin{algorithmic}
\Function{$L \leftarrow \texttt{Cholesky\_Cpt}(A)$}
\For{$k \leftarrow 1$ \textbf{to} $N$}
\For{$j \leftarrow 1$ \textbf{to} $d$} \Comment{Computation of $L(k,:,:)$}
\State $S_1 \leftarrow 0$
\For{$l \leftarrow 1$ \textbf{to} $j - 1$}
\State $S_1 \leftarrow S_1 + |L(k,j,l)|^2$
\EndFor
\State $L(k,j,j) \leftarrow \sqrt{A(k,j,j) - S_1}$
\EndFor
\For{$i \leftarrow j + 1$ \textbf{to} $d$}
\State $S_2 \leftarrow 0$
\For{$l \leftarrow 1$ \textbf{to} $j - 1$}
\State $S_2 \leftarrow S_2 + L(k,i,l) \cdot \texttt{Conj}(L(k,j,l))$
\EndFor
\State $L(k,i,j) \leftarrow (B(k,i,j) - S_2) / L(k,j,j)$
\EndFor
\EndFor
\EndFunction
\end{algorithmic}
\end{algorithm}
Algorithm 32  Function Cholesky_Vec, returns cholesky factorizations of $A_k$ matrices (vectorized)

1: Function L ← Cholesky_Vec ($A$)
2: $L ← \text{Zeros}(N, d, d)$
3: for $j ← 1$ to $d$ do
4:   $S_j ← \text{Zeros}(N)$
5:   for $l ← 1$ to $j - 1$ do
6:      $S_j ← S_j + \sqrt{|L(:, j, l)|^2}$
7:   end for
8:   $L(:, j, j) ← \text{sqrt}(|A(:, j, j) - S_j|)$
9:   for $i ← j + 1$ to $d$ do
10:      $S_i ← \text{Zeros}(N)$
11:     for $l ← 1$ to $j - 1$ do
12:        $S_i ← S_i + L(:, i, l) \cdot \text{conj}(L(:, j, l))$
13:     end for
14:     $L(:, i, j) ← (A(:, i, j) - S_i) / L(:, j, j)$
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 $(\mathcal{M}_d(K))^N$ 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[Nat]. 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 $(\mathcal{M}_{d,d}(K))^N$ 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 $A \in \mathcal{M}_d(C)$ (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 $P \in \mathcal{M}_d(R)$, a lower triangular matrix $L \in \mathcal{M}_d(C)$ with unit diagonal and an upper triangular matrix $U \in \mathcal{M}_d(C)$ such that

$$PA = LU.$$ (23)

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

The object of this section is to describe a vectorized version of these two algorithms apply to 3D-arrays. More precisely, let $A \in (\mathcal{M}_{d,d}(K))^N$, we want to compute the three 3D-arrays $P$, $L$ and $U$ in $(\mathcal{M}_{d,d}(K))^N$ such that, for all
Table 17: Computational times in seconds of \texttt{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.

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

(a) Function \texttt{Cholesky\_Cpt}

(b) Function \texttt{Cholesky\_Mat}

<table>
<thead>
<tr>
<th>$N$</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 000</td>
<td>0.002(s)</td>
<td>0.001(s)</td>
<td>0.001(s)</td>
</tr>
<tr>
<td>40 000</td>
<td>0.002(s)</td>
<td>0.001(s)</td>
<td>0.003(s)</td>
</tr>
<tr>
<td>60 000</td>
<td>0.002(s)</td>
<td>0.002(s)</td>
<td>0.004(s)</td>
</tr>
<tr>
<td>80 000</td>
<td>0.003(s)</td>
<td>0.003(s)</td>
<td>0.005(s)</td>
</tr>
<tr>
<td>100 000</td>
<td>0.004(s)</td>
<td>0.004(s)</td>
<td>0.007(s)</td>
</tr>
</tbody>
</table>

(c) Function \texttt{Cholesky\_Vec}

Table 18: Computational times in seconds of the \texttt{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.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Matlab</th>
<th>Matlab(*)</th>
<th>Octave</th>
<th>Python</th>
<th>Python(Nat)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.009(s)</td>
<td>0.010(s)</td>
<td>0.008(s)</td>
<td>0.019(s)</td>
<td>0.018(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.016(s)</td>
<td>0.019(s)</td>
<td>0.016(s)</td>
<td>0.043(s)</td>
<td>0.036(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.025(s)</td>
<td>0.031(s)</td>
<td>0.035(s)</td>
<td>0.074(s)</td>
<td>0.059(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.033(s)</td>
<td>0.042(s)</td>
<td>0.047(s)</td>
<td>0.106(s)</td>
<td>0.089(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.042(s)</td>
<td>0.054(s)</td>
<td>0.061(s)</td>
<td>0.129(s)</td>
<td>0.110(s)</td>
</tr>
<tr>
<td>5 000 000</td>
<td>0.329(s)</td>
<td>0.445(s)</td>
<td>0.523(s)</td>
<td>0.809(s)</td>
<td>0.542(s)</td>
</tr>
<tr>
<td>10 000 000</td>
<td>0.655(s)</td>
<td>0.904(s)</td>
<td>1.036(s)</td>
<td>1.615(s)</td>
<td>1.079(s)</td>
</tr>
</tbody>
</table>

Table 19: Function \texttt{Cholesky\_Vec} with $\mathbf{A} \in (\mathcal{M}_{3,3}(\mathbb{K}))^N$ under Matlab 2018a: effect of multithreaded on \texttt{cpumtimes}
Figure 5: Computational times in seconds of `cholesky_vec` function with $A \in (\mathcal{M}_{d,d}(K))^N$ with $N = 10^6$ and $d \in [2, 10]$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.
$k \in [1, N]$, a LU factorization with partial pivoting of the matrix $A_k = A(k, \cdot, \cdot)$ is given by

$$P_k A_k = L_k U_k$$

where $P_k = P(k, \cdot, \cdot)$, $L_k = L(k, \cdot, \cdot)$ and $U_k = U(k, \cdot, \cdot)$.

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

---

**Données** : $A$ : matrix in $M_{d}(K)$.

**Résultat** : $P$ : permutation matrix in $M_{d}(K)$,

$L$ : lower triangular matrix in $M_{d}(K)$

avec $L(i, i) = 1$, $\forall i \in [1, d]$,

$U$ : upper triangular matrix in $M_{d}(K)$.

---

1: Function $[P, L, U] \leftarrow \text{PALU} (A)$
2: $P \leftarrow \text{Eye}(d)$, $L \leftarrow \text{Eye}(d)$, $U \leftarrow A$
3: for $i \leftarrow 1$ to $d$ do
4: $\mu \leftarrow \text{Argmax}([|U(i : d, i)|] + (i - 1))$
5: if $|U(\mu, i)| > \epsilon$ then
6: if $\mu \neq i$ then $\quad$ Permute rows $i$ and $\mu$
7: $U(i, i : d) \leftarrow U(\mu, i : d)$ $\quad$ Only columns $i$ to $d$
8: $L(i, 1 : i - 1) \leftarrow L(\mu, 1 : i - 1)$ $\quad$ Only columns 1 to $i - 1$
9: $P(i, :) \leftarrow P(\mu, :)$ $\quad$ All columns
10: end if
11: end if
12: for $j \leftarrow i + 1$ to $d$ do
13: $L(j, i) \leftarrow U(j, i)/U(i, i)$
14: $L(j, i : d) \leftarrow U(j, i : d) - L(j, i) \times U(i, i : nd)$
15: end for
16: end for
17: end Function
Algorithm 34 Function \texttt{pLUinplace} inplace computation of the LU factorization with partial pivoting of a matrix \( A \) such that \( PA = LU \).

\begin{enumerate}
    \item \textbf{Données :} \( A \) : matrix in \( \mathcal{M}_n(K) \).
    \item \textbf{Résultat :} \( p \) : rows permutation index
    \item \( A \) : the modified matrix such that ...
\end{enumerate}

1: Define the function \([p, A] \leftarrow \texttt{pLUinplace} \ (A)\)
2: \( p \leftarrow 1 : n \)
3: for \( i \leftarrow 1 \) to \( n - 1 \) do
4: \( \mu \leftarrow \texttt{argmax} \left(A(i : n, i) + (i - 1)\right) \)
5: if \( |A(\mu, i)| > \epsilon \) then
6: \( \text{perms rows } i \text{ and } \mu \)
7: \( p(i) \leftarrow p(\mu) \)
8: end if
9: \( I \leftarrow i + 1 : n \)
10: \( \delta(I, i) \leftarrow \delta(I, i)/\delta(i, i) \)
11: \( \delta(I, I) \leftarrow \delta(I, I) - \delta(I, i) * \delta(i, I) \)
12: end if
13: end for
14: end Function

Unlike the cholesky decomposition, there is currently no Numpy or 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 \( A \in (\mathcal{M}_{d,d}(K))^N \). This code use on each matrix \( A(k, :) \) the function \texttt{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 \texttt{PALU-Mat} computes all LU factorizations with partial pivoting of a 3D-array \( A \) such that \( P_k A_k = L_k U_k \)

\begin{enumerate}
    \item \textbf{Input :} \( A \) : in \( (\mathcal{M}_{d,d}(K))^N \).
    \item \textbf{Output :} \( P \) : permutation matrices in \( (\mathcal{M}_{d,d}(K))^N \),
    \item \( L \) : lower triangular matrices in \( (\mathcal{M}_{d,d}(K))^N \) avec \( L_k(i, i) = 1, \forall i \in [1, d] \),
    \item \( U \) : upper triangular matrices in \( (\mathcal{M}_{d,d}(K))^N \).
\end{enumerate}

1: Define the function \([P, L, U] \leftarrow \texttt{PALU-Mat} \ (A)\)
2: for \( k \leftarrow 1 \) to \( N \) do
3: \( [P(k, :, :), L(k, :, :), U(k, :, :)] \leftarrow \texttt{PALU}(A(k, :, :)) \)
4: end for
5: end Function
Algorithm 36 Function \texttt{PALU\_Cpt} computes all LU factorizations with partial pivoting of a 3D-array \( \mathbf{A} \) such that \( \mathbf{P}_k \mathbf{A}_k = \mathbf{L}_k \mathbf{U}_k \)

1: Function \( [\mathbf{P}, \mathbf{L}, \mathbf{U}] \leftarrow \text{PALU\_Cpt} \left( \mathbf{A} \right) \)
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 \( \Rightarrow \) Permutes rows \( i \) and \( \mu \)
7: \( \mathbf{U}(k, i : d) \leftrightarrow \mathbf{U}(k, \mu, i : d) \) \( \Rightarrow \) Only columns \( i \) to \( d \)
8: \( \mathbf{L}(k, i : 1 : i - 1) \leftrightarrow \mathbf{L}(k, \mu, 1 : i - 1) \) \( \Rightarrow \) Only columns \( 1 \) to \( i - 1 \)
9: \( \mathbf{P}(k, i, :) \leftrightarrow \mathbf{P}(k, \mu, :) \) \( \Rightarrow \) All columns
10: end if
11: for \( j \leftarrow i + 1 \) to \( d \) do \( \Rightarrow \) Elimination
12: \( \mathbf{L}(k, j, i) \leftarrow \mathbf{U}(k, j, i)/\mathbf{U}(k, i, i) \)
13: \( \mathbf{L}(k, j, i : d) \leftarrow \mathbf{U}(k, j, i : d) - \mathbf{L}(k, j, i) \ast \mathbf{U}(k, i, i : nd) \)
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 \( i \in [1, d] \), finding all pivoting index for each matrix on column \( i \) and rows \( i : d \) on a not vectorized way is:

\[
\mu(k) \leftarrow \text{Argmax}(|\mathbf{U}(k, i : d, i)|) + (i - 1), \quad \forall k \in [1, N]
\]

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

\[
\mu \leftarrow \text{Argmax}(|\mathbf{U}(; i : d, i)|, 2) + (i - 1).
\]

Permutation at step \( i \) for \( \mathbf{P} \) is

\[
\mathbf{P}(k, i, :) \leftrightarrow \mathbf{P}(k, \mu(k), :), \quad \forall k \in [1, N]
\]

This can be written as

\[
\text{for } j \leftarrow 1 \text{ to } d \text{ do} \\
\quad \mathbf{P}(k, i, j) \leftrightarrow \mathbf{P}(k, \mu(k), j), \quad \forall k \in [1, N] \\
\text{end for}
\]

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.

\[
J \leftarrow \text{ONES} \left( 1, N \right) \\
\text{for } j \leftarrow 1 \text{ to } d \text{ do} \\
\quad I_1 \leftarrow \text{Sub2Ind} \left( \left[ N, d, d \right], 1 : N, \mu, j \ast J \right) \\
\quad I_2 \leftarrow \text{Sub2Ind} \left( \left[ N, d, d \right], 1 : N, i \ast J, j \ast J \right) \\
\quad \mathbf{P}(I_1) \leftrightarrow \mathbf{P}(I_2) \\
\text{end for}
\]
Permutation rows for $L$ and $U$ are obtained in a similar way.

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

$$K \leftarrow |U(:, i, i)| > \epsilon$$

for $j \leftarrow i + 1$ to $d$ do

$L(K, j, i) \leftarrow U(K, j, i) \cdot U(K, i, i)$

$L(K, j, i : d) \leftarrow U(K, j, i : d) - U(K, i, i : d)$

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 $A$ such that $P_k A_k = L_k U_k$

1: Function $[P, L, U] \leftarrow$ PALU_Vec ($A$)
2: $K \leftarrow 1 : N$
3: $J \leftarrow \text{ONES}(1, N)$
4: for $i \leftarrow 1$ to $d$ do
5: $\mu \leftarrow \text{ARGMAX}([U(:, i : d, i)], 2) + (i - 1)$
6: for $j \leftarrow 1$ to $i - 1$ do
7: $I_1 \leftarrow \text{SUB2IND}([N, d, d], K, \mu, j \ast J)$
8: $I_2 \leftarrow \text{SUB2IND}([N, d, d], K, i \ast J, j \ast J)$
9: $L(I_1) \leftarrow L(I_2)$
10: $P(I_1) \leftarrow P(I_2)$
11: end for
12: for $j \leftarrow i + 1$ to $d$ do
13: $I_1 \leftarrow \text{SUB2IND}([N, d, d], K, \mu, j \ast J)$
14: $I_2 \leftarrow \text{SUB2IND}([N, d, d], K, i \ast J, j \ast J)$
15: $U(I_1) \leftarrow U(I_2)$
16: $P(I_1) \leftarrow P(I_2)$
17: end for
18: $Kidx \leftarrow |U(:, i, i)| > \epsilon$
19: for $j \leftarrow i + 1$ to $d$ do
20: $L(Kidx, j, i) \leftarrow U(Kidx, j, i) \cdot U(Kidx, i, i)$
21: $L(Kidx, j, i : d) \leftarrow U(Kidx, j, i : d) - L(Kidx, i, i : d)$
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 $(\mathcal{M}_{3,3}(K))^N$. 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 $(\mathcal{M}_{d,d}(K))^N$ for $N = 10^8$ and $d \in [2, 10]$.  

34
Table 20: Computational times in seconds of \(*\)PALU* functions with \(A \in (M_{3,3}(K))^N\) for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

<table>
<thead>
<tr>
<th>(N)</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>3.47(s)</td>
<td>42.20(s)</td>
<td>6.13(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>7.001(s)</td>
<td>84.58(s)</td>
<td>12.48(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>10.471(s)</td>
<td>127.051(s)</td>
<td>18.614(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>13.9056(s)</td>
<td>169.286(s)</td>
<td>25.095(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>17.375(s)</td>
<td>209.688(s)</td>
<td>31.455(s)</td>
</tr>
</tbody>
</table>

(a) Function \(*\)PALU* _Cpr_

<table>
<thead>
<tr>
<th>(N)</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.082(s)</td>
<td>0.075(s)</td>
<td>0.127(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.168(s)</td>
<td>0.159(s)</td>
<td>0.271(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.253(s)</td>
<td>0.265(s)</td>
<td>0.446(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.342(s)</td>
<td>0.384(s)</td>
<td>0.627(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.431(s)</td>
<td>0.467(s)</td>
<td>0.811(s)</td>
</tr>
</tbody>
</table>

(b) Function \(*\)PALU* _Mat_

<table>
<thead>
<tr>
<th>(N)</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.090(s)</td>
<td>0.091(s)</td>
<td>0.076(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.209(s)</td>
<td>0.207(s)</td>
<td>0.184(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.251(s)</td>
<td>0.308(s)</td>
<td>0.268(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.339(s)</td>
<td>0.426(s)</td>
<td>0.388(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.447(s)</td>
<td>0.535(s)</td>
<td>0.480(s)</td>
</tr>
<tr>
<td>5 000 000</td>
<td>3.798(s)</td>
<td>4.428(s)</td>
<td>3.977(s)</td>
</tr>
<tr>
<td>10 000 000</td>
<td>7.374(s)</td>
<td>8.934(s)</td>
<td>8.138(s)</td>
</tr>
</tbody>
</table>

(c) Function \(*\)PALU* _Vec_

Table 21: Computational times in seconds of \(*\)PALU* _Vec_ functions with \(A \in (M_{3,3}(K))^N\) for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

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

Table 22: Function \(*\)PALU* _Vec_ with \(A \in (M_{3,3}(K))^N\) under Matlab 2018a: effect of multithreaded on cpumtimes
### Figure 6

Computational times in seconds of the `PALU_VEC` function with $\mathbb{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$ with $N = 10^6$ and $d \in [2, 10]$ 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 $\mathbb{A} \in (\mathcal{M}_{d,d}(\mathbb{K}))^N$. Before vectorizing, one have to write the complete code without using this function. This is done in Algorithm 39.

**Algorithm 38** Function `pLUIntplace_Mat` computes all LU factorizations with partial pivoting of a 3D-array $\mathbb{A}$ such that $P_k A_k = L_k U_k$.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{A}$</td>
<td>$p$</td>
<td>permutation index array in $(\mathcal{M}_d(\mathbb{K}))^N$.</td>
</tr>
<tr>
<td>$\mathbb{A}$</td>
<td></td>
<td>the modified 3D-array</td>
</tr>
</tbody>
</table>

```python
1: Function $[p, \mathbb{A}] \leftarrow pLUIntplace_Mat(\mathbb{A})$
2: for $k \leftarrow 1$ to $N$ do
3: \hspace{1em} $[p(k,:), \mathbb{A}(k,:,:)] \leftarrow pLUIntplace(\mathbb{A}(k,:,))$
4: end for
5: end Function
```
Algorithm 39 Function \texttt{pLUinplace\_Cpt} computes all LU factorizations with partial pivoting of a 3D-array \( \text{A} \) such that \( \mathbb{P}_k \text{A}_k = \text{L}_k \text{U}_k \)

1: Function \( [\mathbb{P}, \text{A}] \leftarrow \text{pLUinplace\_Cpt} \left( \text{A} \right) \)
2: \( \mathbb{P} \leftarrow \text{RepTile}(1 : n, N, 1) \)
3: for \( k \leftarrow 1 \) to \( N \) do
4: for \( i \leftarrow 1 \) to \( n - 1 \) do
5: \( \mu \leftarrow \text{Argmax}(\text{A}(k, i : n, i)) + (i - 1) \)
6: if \( |\text{A}(k, \mu, i)| > \epsilon \) then
7: if \( \mu \neq i \) then \( \Rightarrow \text{Permutates rows } i \text{ and } \mu \)
8: \( \text{A}(k, i, :) \leftarrow \text{A}(k, \mu, :) \)
9: \( \mathbb{P}(k, i) \leftarrow \mathbb{P}(k, \mu) \)
10: end if
11: end for
12: end for
13: end Function

The complete vectorized code is given by \texttt{pLUinplace\_Vec} function in Algorithm 40 and is close to the \texttt{PALU\_Vec} function given in Algorithm 37.

Algorithm 40 Function \texttt{pLUinplace\_Vec} computes all LU factorizations with partial pivoting of a 3D-array \( \text{A} \) such that \( \mathbb{P}_k \text{A}_k = \text{L}_k \text{U}_k \)

1: Function \( [\mathbb{P}, \text{A}] \leftarrow \text{pLUinplace\_Vec} \left( \text{A} \right) \)
2: \( \mathbb{P} \leftarrow \text{RepTile}(1 : d, N, 1) \)
3: \( K \leftarrow 1 : N \)
4: \( J \leftarrow \text{Ones}(1, N) \)
5: \( \text{for } i \leftarrow 1 \text{ to } n - 1 \) do
6: \( \mu \leftarrow \text{Argmax}(|\text{U}(; i : d, i)|, 2) + (i - 1) \)
7: \( \text{for } j \leftarrow 1 \text{ to } d \) do
8: \( \text{I}_1 \leftarrow \text{Sub2Ind}([N, d, d], K, \mu, j : J) \)
9: \( \text{I}_2 \leftarrow \text{Sub2Ind}([N, d, d], K, i : J, j : J) \)
10: \( \text{A}(\text{I}_1) \leftarrow \text{A}(\text{I}_2) \)
11: end for
12: \( \text{I}_1 \leftarrow \text{Sub2Ind}([N, d], K, \mu) \)
13: \( \text{I}_2 \leftarrow \text{Sub2Ind}([N, d], K, i : J) \)
14: \( \mathbb{P}(\text{I}_1) \leftarrow \mathbb{P}(\text{I}_2) \)
15: \( \text{Kidx} \leftarrow |\text{U}(; i, i)| > \epsilon \)
16: \( \text{for } j \leftarrow i + 1 \text{ to } d \) do
17: \( \text{L}(\text{Kidx}, j, i) \leftarrow \text{U}(\text{Kidx}, j, i) / \text{U}(\text{Kidx}, i, i) \)
18: \( \text{L}(\text{Kidx}, j, i : d) \leftarrow \text{U}(\text{Kidx}, j, i : d) - \text{L}(\text{Kidx}, j, i) \cdot \text{U}(\text{Kidx}, i, i : d) \)
19: end for
20: end for
21: end Function

37
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 $(M_{3,3}(K))^N$. 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 $(\mathcal{M}_{d,d}(K))^N$ for $N = 10^6$ and $d \in [2, 10]$.

<table>
<thead>
<tr>
<th>$N$ (\text{Matlab})</th>
<th>Octave</th>
<th>Python</th>
<th>$N$ (\text{Matlab})</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>3.701(s)</td>
<td></td>
<td>200 000</td>
<td>4.108(s)</td>
<td>4.060(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>7.508(s)</td>
<td></td>
<td>400 000</td>
<td>8.376(s)</td>
<td>8.086(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>11.289(s)</td>
<td></td>
<td>600 000</td>
<td>12.572(s)</td>
<td>11.888(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>14.960(s)</td>
<td></td>
<td>800 000</td>
<td>16.668(s)</td>
<td>14.695(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>18.616(s)</td>
<td></td>
<td>1 000 000</td>
<td>20.859(s)</td>
<td>18.274(s)</td>
</tr>
</tbody>
</table>

(a) Function `PLUinplace_Cpt`  
(b) Function `PLUinplace_Mat`  
(c) Function `PLUinplace_Vec`

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

<table>
<thead>
<tr>
<th>$N$ (\text{Matlab})</th>
<th>Matlab(*)</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.071(s)</td>
<td>0.079(s)</td>
<td>0.083(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.140(s)</td>
<td>0.169(s)</td>
<td>0.169(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.198(s)</td>
<td>0.265(s)</td>
<td>0.299(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.287(s)</td>
<td>0.342(s)</td>
<td>0.420(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.356(s)</td>
<td>0.500(s)</td>
<td>0.500(s)</td>
</tr>
<tr>
<td>5 000 000</td>
<td>2.897(s)</td>
<td>3.784(s)</td>
<td>4.335(s)</td>
</tr>
<tr>
<td>10 000 000</td>
<td>5.628(s)</td>
<td>7.559(s)</td>
<td>8.489(s)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$N$</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>3 Threads</th>
<th>4 Threads</th>
<th>5 Threads</th>
<th>6 Threads</th>
<th>8 Threads</th>
<th>10 Threads</th>
<th>12 Threads</th>
<th>14 Threads</th>
<th>16 Threads</th>
<th>24 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.060(s)</td>
<td>0.060(s)</td>
<td>0.061(s)</td>
<td>0.060(s)</td>
<td>0.060(s)</td>
<td>0.060(s)</td>
<td>0.060(s)</td>
<td>0.060(s)</td>
<td>0.060(s)</td>
<td>0.060(s)</td>
<td>0.060(s)</td>
<td>0.060(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.150(s)</td>
<td>0.150(s)</td>
<td>0.150(s)</td>
<td>0.150(s)</td>
<td>0.149(s)</td>
<td>0.149(s)</td>
<td>0.149(s)</td>
<td>0.149(s)</td>
<td>0.149(s)</td>
<td>0.149(s)</td>
<td>0.149(s)</td>
<td>0.149(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.250(s)</td>
<td>0.250(s)</td>
<td>0.250(s)</td>
<td>0.250(s)</td>
<td>0.250(s)</td>
<td>0.250(s)</td>
<td>0.250(s)</td>
<td>0.250(s)</td>
<td>0.250(s)</td>
<td>0.250(s)</td>
<td>0.250(s)</td>
<td>0.250(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.350(s)</td>
<td>0.350(s)</td>
<td>0.350(s)</td>
<td>0.350(s)</td>
<td>0.350(s)</td>
<td>0.350(s)</td>
<td>0.350(s)</td>
<td>0.350(s)</td>
<td>0.350(s)</td>
<td>0.350(s)</td>
<td>0.350(s)</td>
<td>0.350(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.450(s)</td>
<td>0.450(s)</td>
<td>0.450(s)</td>
<td>0.450(s)</td>
<td>0.450(s)</td>
<td>0.450(s)</td>
<td>0.450(s)</td>
<td>0.450(s)</td>
<td>0.450(s)</td>
<td>0.450(s)</td>
<td>0.450(s)</td>
<td>0.450(s)</td>
</tr>
<tr>
<td>5 000 000</td>
<td>0.750(s)</td>
<td>0.750(s)</td>
<td>0.750(s)</td>
<td>0.750(s)</td>
<td>0.750(s)</td>
<td>0.750(s)</td>
<td>0.750(s)</td>
<td>0.750(s)</td>
<td>0.750(s)</td>
<td>0.750(s)</td>
<td>0.750(s)</td>
<td>0.750(s)</td>
</tr>
<tr>
<td>10 000 000</td>
<td>7.450(s)</td>
<td>7.450(s)</td>
<td>7.450(s)</td>
<td>7.450(s)</td>
<td>7.450(s)</td>
<td>7.450(s)</td>
<td>7.450(s)</td>
<td>7.450(s)</td>
<td>7.450(s)</td>
<td>7.450(s)</td>
<td>7.450(s)</td>
<td>7.450(s)</td>
</tr>
</tbody>
</table>

Table 25: Function `PLUinplace_Vec` with $A \in (M_{3,3}(K))^N$ under Matlab 2018a: effect of multithreaded on cputimes
Figure 7: Computational times in seconds of \texttt{pLUinplace.\_Vec} function with \( A \in (\mathcal{M}_{d,d}(\mathbb{K}))^N \) with \( N = 10^6 \) and \( d \in [2, 10] \) 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

\[
A \mathbf{x} = B
\]
as described in section 4.2.3

5.1 Using Cholesky factorization

Let \( A \in \mathcal{M}_{d,d}(\mathbb{K}) \) 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 \( A = \mathbb{L}\mathbb{L}^* \). So, to solve the equation

\[
A \mathbf{x} = B
\]

one just have to solve the two triangular systems

\[
\mathbb{L} \mathbf{y} = \mathbf{b} \quad \text{then} \quad \mathbb{L}^* \mathbf{x} = \mathbf{y}.
\]

For a symmetric positive definite 3D-array we immediately have the \texttt{Lin-\_solve\_Cholesky\_\_Vec} vectorized function given in Algorithm 41. This function
uses the vectorized functions \( \text{Cholesky\_Vec} \), \( \text{LinsolveTriL\_Vec} \), \( \text{LinsolveTriU\_Vec} \) and \( \text{ctranspose\_Vec} \) respectively given in Algorithms 32, 28, 28 and ?? . In Table 26, computational times in second with \( A \in (M_{3,3}(K))^N \) and \( B \in (M_{3,1}(K))^N \) under Matlab, Octave and Python are given for \( N \) values up to \( 10^7 \). In Figure 8 the computation time in second of the \( \text{LinsolveCholesky\_Vec} \) with \( A \in (M_{d,d}(K))^N \) and \( B \in (M_{d,1}(K))^N \) for \( N = 10^6 \) and \( d \in [2,10] \) is represented.

**Algorithm 41** Function \( \text{LinsolveCholesky\_Vec} \), solves equation \( AX = B \) where \( A \) is a symmetric positive definite 3D-array(vectorized)

\[
\begin{align*}
\text{Function } & X \leftarrow \text{LinsolveCholesky\_Vec}(A, B) \\
& L \leftarrow \text{Cholesky\_Vec}(A) \\
& Y \leftarrow \text{LinsolveTriL\_Vec}(L, B) \\
& X \leftarrow \text{LinsolveTriU\_Vec}(\text{ctranspose\_Vec}(L), Y)
\end{align*}
\]

<table>
<thead>
<tr>
<th>( N )</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.024(s)</td>
<td>0.023(s)</td>
<td>0.037(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.044(s)</td>
<td>0.041(s)</td>
<td>0.080(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.071(s)</td>
<td>0.084(s)</td>
<td>0.132(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.092(s)</td>
<td>0.111(s)</td>
<td>0.176(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.119(s)</td>
<td>0.142(s)</td>
<td>0.221(s)</td>
</tr>
<tr>
<td>5 000 000</td>
<td>0.786(s)</td>
<td>1.300(s)</td>
<td>1.441(s)</td>
</tr>
<tr>
<td>10 000 000</td>
<td>1.789(s)</td>
<td>2.589(s)</td>
<td>2.868(s)</td>
</tr>
</tbody>
</table>

Table 26: Computational times in seconds of the \( \text{LinsolveCholesky\_Vec} \) function with \( A \in (M_{3,3}(K))^N \) and \( B \in (M_{3,1}(K))^N \) for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

### 5.2 Using LU factorization with partial pivoting

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

\[
AX = B
\]

one just have to solve the two triangular systems

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

For a regular 3D-array we immediately deduce the \( \text{LinsolvePALU\_Vec} \) vectorized function given in vectorized Algorithm 42. This function uses the vectorized functions \( \text{PALU\_Vec} \), \( \text{LinsolveTriL\_Vec} \) and \( \text{LinsolveTriU\_Vec} \) respectively given in Algorithms 37, 24 and 28. In Table 27, computational times in second with \( A \in (M_{3,3}(K))^N \) and \( B \in (M_{3,1}(K))^N \) 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 \text{numpy.linalg.solve} function that we
Figure 8: Computational times in seconds of \texttt{LINSOLVE_CHOLESKY_VEC} function with $A \in (M_{d,d}(K))^N$ with $N = 10^6$ and $d \in [2,10]$ 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 \texttt{LinsolvePALU\_Vec} with $\mathbf{A} \in (M_{d,d}(K))^N$ and $\mathbf{B} \in (M_{d,1}(K))^N$ for $N = 10^6$ and $d \in [2, 10]$ is represented.

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

\begin{verbatim}
    Function $\mathbf{X} \leftarrow \texttt{LinsolvePALU\_Vec}(\mathbf{A}, \mathbf{B})$
    $[\mathbf{P}, \mathbf{L}, \mathbf{U}] \leftarrow \texttt{PALU\_VEC}(\mathbf{A})$
    $\mathbf{Y} \leftarrow \texttt{LinsolveTriL\_Vec}(\mathbf{L}, \texttt{mtimes\_Vec}(\mathbf{P}, \mathbf{B}))$
    $\mathbf{X} \leftarrow \texttt{LinsolveTriU\_Vec}(\mathbf{U}, \mathbf{Y})$
end Function
\end{verbatim}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
$N$ & Matlab & Octave & Python & Python[Nat] \\
\hline
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) \\
\hline
\end{tabular}
\caption{Computational times in seconds of the \texttt{LinsolvePALU\_Vec} function with $\mathbf{A} \in (M_{3,3}(K))^N$ and $\mathbf{B} \in (M_{3,1}(K))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. The last column is for the native python function \texttt{numpy.linalg.solve}.}
\end{table}
Figure 9: Computational times in seconds of \texttt{LINSOLVE\_PALU\_VEC} function with $A \in (\mathcal{M}_{d,d}(K))^N$ with $N = 10^6$ and $d \in [2, 10]$ 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 \( \text{Det} \) for a matrix in \( \mathcal{M}_d(K) \), 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 \texttt{numpy.linalg.det} Python function of the Numpy package can performed directly this operation.

Algorithm 43 Function \( \text{Det}_M \), returns determinants of a 3D-array (not vectorized)

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

Output \( D \) : in \( \mathbb{R}^N \)

Function \( D \leftarrow \text{Det}_M (A) \)

for \( k \leftarrow 1 \) to \( N \) do

\( D(k) \leftarrow \text{Det} (A(k,:,::)) \)

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

\[
\det \mathbb{B} = \sum_{j=1}^{n} b_{i,j} C_{i,j} \overset{\text{def}}{=} \det_1 \mathbb{B}
\]

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 \( \text{DetLap} \) using the formula (24).

For a 3D-array \( \mathbb{A} \) in \( (\mathcal{M}_{d,d}(K))^N \), we deduce the two non-vectorized function using the \( \text{DetLap} \) function given in Algorithms 45 and 46. From the last one, we easily obtain the vectorized function \( \text{DetLap}_\text{Vec} \) given in Algorithm 47.
To overcome, instead of creating a new 3D array from 
the function, we only create a row and column indices as 1D arrays. This is the expanded with respect to the 1st row.

Algorithm 44 Function \texttt{DetLap}, returns determinant of the matrix \(B\) by using Laplace formula \((24)\) expanded with respect to the 1st row.
\begin{verbatim}
Input \(B\) : a \(d\)-by-\(d\) matrix 
Output \(r\) : the scalar \(\text{det}(B)\).
\end{verbatim}
\begin{algorithm}
\caption{Function \texttt{DetLap}, returns determinant of the matrix \(B\) by using Laplace formula \((24)\) expanded with respect to the 1st row.}
\begin{algorithmic}
\Function{\texttt{DetLap}}{(B)}
\State \textbf{if} \texttt{d} \texttt{==} 1 \textbf{then}
\State \texttt{r} \texttt{=} \texttt{B(1,1)}
\Else
\State \texttt{r} \texttt{=} 0
\For{\texttt{j} \texttt{=} 1 \texttt{to} \texttt{d} \texttt{do}}
\State \texttt{r} \texttt{=} \texttt{r} \texttt{+} \texttt{(-1)}^{\texttt{j}} \texttt{+ 1} \texttt{* B(1,}\texttt{j})
\EndFor
\State \texttt{DetLap(B(2:d,}\texttt{1:j-1,}\texttt{j+1:d}))
\EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}

Algorithm 45 Function \texttt{DetLap_Mat}, returns determinants of a 3D array (not vectorized).
\begin{algorithm}
\caption{Function \texttt{DetLap_Mat}, returns determinants of a 3D array (not vectorized)}
\begin{algorithmic}
\Function{\texttt{DetLap_Mat}}{(A)}
\For{\texttt{k} \texttt{=} 1 \texttt{to} \texttt{N} \texttt{do}}
\State \texttt{D(k)} \texttt{=} \texttt{DetLap(A(k,:,:))}
\EndFor
\EndFunction
\end{algorithmic}
\end{algorithm}

Algorithm 46 Function \texttt{DetLap_Cpt}, returns determinants of a 3D array in \((M_{d,2}(K))^N\) (not vectorized).
\begin{algorithm}
\caption{Function \texttt{DetLap_Cpt}, returns determinants of a 3D array in \((M_{d,2}(K))^N\) (not vectorized).}
\begin{algorithmic}
\Function{\texttt{DetLap_Cpt}}{(A)}
\If{\texttt{d}\texttt{==} 1}
\State \texttt{D} \texttt{=} \texttt{A}
\Else
\State \texttt{D} \texttt{=} \texttt{Zeros(N,1)}
\For{\texttt{k} \texttt{=} 1 \texttt{to} \texttt{N} \texttt{do}}
\For{\texttt{j} \texttt{=} 1 \texttt{to} \texttt{d} \texttt{do}}
\State \texttt{D(k)} \texttt{=} \texttt{D(k)} \texttt{+ (-1)}^{\texttt{j}} \texttt{+ 1} \texttt{* A(k,}\texttt{j},\texttt{1:j})
\EndFor
\EndFor
\State \texttt{DetLap(A(k,2:d,}\texttt{1:j-1,}\texttt{j+1:d}))
\EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}

Algorithm 47 Function \texttt{DetLap_Vec}, returns determinants of a 3D array in \((M_{d,2}(K))^N\) (not vectorized).
\begin{algorithm}
\caption{Function \texttt{DetLap_Vec}, returns determinants of a 3D array in \((M_{d,2}(K))^N\) (not vectorized).}
\begin{algorithmic}
\Function{\texttt{DetLap_Vec}}{(A)}
\If{\texttt{d}\texttt{==} 1}
\State \texttt{D} \texttt{=} \texttt{A}
\Else
\State \texttt{D} \texttt{=} \texttt{Zeros(N,1)}
\For{\texttt{j} \texttt{=} 1 \texttt{to} \texttt{d} \texttt{do}}
\State \texttt{D} \texttt{=} \texttt{D} \texttt{+ (-1)}^{\texttt{j}} \texttt{+ 1} \texttt{* A(}\texttt{k},\texttt{1:j})
\State \texttt{DetLap_Vec(A(k,2:d,}\texttt{1:j-1,}\texttt{j+1:d}))
\EndFor
\EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}

The major disadvantage of the Algorithm 47 is that it is memory consuming. To overcome, instead of creating a new 3D array from \(A\) 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 \texttt{DetLapMat}, returns determinants of \(A\) matrices by using Laplace formula \((24)\) expanded with respect to the 1st row (vectorized, recursive and memory safe).
\begin{algorithm}
\caption{Function \texttt{DetLapMat}, returns determinants of \(A\) matrices by using Laplace formula \((24)\) expanded with respect to the 1st row (vectorized, recursive and memory safe).}
\begin{algorithmic}
\Function{\texttt{DetLapMat}}{(A, I, J)}
\If{\texttt{I} \texttt{==} \emptyset \texttt{and} \texttt{J} \texttt{==} \emptyset}
\State \texttt{m} \texttt{=} \texttt{d}
\State \texttt{I} \texttt{=} 1 \texttt{:} \texttt{d}, \texttt{J} \texttt{=} 1 \texttt{:} \texttt{d}
\Else
\State \texttt{m} \texttt{=} \texttt{Len(I)}
\EndIf
\If{\texttt{m} \texttt{==} 1}
\State \texttt{D} \texttt{=} \texttt{A(}\texttt{I(1),J(1))}
\Else
\State \texttt{D} \texttt{=} \texttt{Zeros(1,\texttt{N})}
\For{\texttt{j} \texttt{=} 1 \texttt{to} \texttt{m} \texttt{do}}
\State \texttt{D} \texttt{=} \texttt{D} \texttt{+ (-1)}^{\texttt{j}} \texttt{+ 1} \texttt{* A(}\texttt{I(1),J(1:j))}
\State \texttt{DetLapMat(A,}\texttt{I[2:m],J[1:j-1,\texttt{j+1:m]}})
\EndFor
\EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}

45
### Function `DetLap_Crt`

<table>
<thead>
<tr>
<th>N</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>6.723(s)</td>
<td>60.282(s)</td>
<td>38.974(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>13.522(s)</td>
<td>121.649(s)</td>
<td>78.228(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>20.281(s)</td>
<td>181.437(s)</td>
<td>117.255(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>27.172(s)</td>
<td>239.521(s)</td>
<td>156.981(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>35.669(s)</td>
<td>300.370(s)</td>
<td>196.141(s)</td>
</tr>
</tbody>
</table>

### Function `DetLap_Mat`

<table>
<thead>
<tr>
<th>N</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.006(s)</td>
<td>0.007(s)</td>
<td>0.018(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.016(s)</td>
<td>0.015(s)</td>
<td>0.038(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.024(s)</td>
<td>0.023(s)</td>
<td>0.058(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.033(s)</td>
<td>0.031(s)</td>
<td>0.078(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.043(s)</td>
<td>0.058(s)</td>
<td>0.152(s)</td>
</tr>
<tr>
<td>5 000 000</td>
<td>0.428(s)</td>
<td>0.549(s)</td>
<td>0.826(s)</td>
</tr>
<tr>
<td>10 000 000</td>
<td>0.842(s)</td>
<td>1.067(s)</td>
<td>1.651(s)</td>
</tr>
</tbody>
</table>

### Function `DetLap_Vec`

<table>
<thead>
<tr>
<th>N</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.009(s)</td>
<td>0.007(s)</td>
<td>0.018(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.016(s)</td>
<td>0.015(s)</td>
<td>0.038(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.024(s)</td>
<td>0.023(s)</td>
<td>0.058(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.033(s)</td>
<td>0.031(s)</td>
<td>0.078(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.043(s)</td>
<td>0.058(s)</td>
<td>0.152(s)</td>
</tr>
<tr>
<td>5 000 000</td>
<td>0.428(s)</td>
<td>0.549(s)</td>
<td>0.826(s)</td>
</tr>
<tr>
<td>10 000 000</td>
<td>0.842(s)</td>
<td>1.067(s)</td>
<td>1.651(s)</td>
</tr>
</tbody>
</table>

### Function `DetLap1dx_Vec`

<table>
<thead>
<tr>
<th>N</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 000</td>
<td>0.009(s)</td>
<td>0.009(s)</td>
<td>0.007(s)</td>
</tr>
<tr>
<td>400 000</td>
<td>0.016(s)</td>
<td>0.018(s)</td>
<td>0.015(s)</td>
</tr>
<tr>
<td>600 000</td>
<td>0.024(s)</td>
<td>0.028(s)</td>
<td>0.023(s)</td>
</tr>
<tr>
<td>800 000</td>
<td>0.033(s)</td>
<td>0.038(s)</td>
<td>0.031(s)</td>
</tr>
<tr>
<td>1 000 000</td>
<td>0.043(s)</td>
<td>0.049(s)</td>
<td>0.058(s)</td>
</tr>
<tr>
<td>5 000 000</td>
<td>0.428(s)</td>
<td>0.492(s)</td>
<td>0.549(s)</td>
</tr>
<tr>
<td>10 000 000</td>
<td>0.842(s)</td>
<td>0.972(s)</td>
<td>1.067(s)</td>
</tr>
</tbody>
</table>

Table 28: Computational times in seconds of `DetLap1dx` functions with $A \in (M_{3,3}(K))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.

### Function `DetLap1dx_Vec`

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

Table 29: Computational times in seconds of `DetLap1dx_Vec` functions with $A \in (M_{3,3}(K))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. The last column is for the native python function `numpy.linalg.det`.

### Function `DetLap1dx_Vec`

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

Table 30: Computational times in seconds of `DetLap1dx_Vec` functions with $A \in (M_{3,3}(K))^N$ 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 \( \text{D} \text{ETL} \text{API} \text{DX} \_ \text{VEC} \) function with \( \mathbf{A} \in (M_{d,d}(K))^N \) with \( N = 10^6 \) and \( d \in [2, 10] \) for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.
\[ PA = LU. \tag{25} \]

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

\[ \det P \det A = \det L \det U. \]

As \( P \) is a permutation matrix we have

\[ \det P = \pm 1. \]

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

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

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

### 6.3 Vectorized algorithm using another expansion

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

\[ \det B = \frac{\det(y^{[1,1]}) \det(y^{[n,n]}) - \det(y^{[n,1]}) \det(y^{[1,n]})}{\det(B^{[2,2]})} \tag{26} \]

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

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

\[ \det B = \frac{\det_2(y^{[1,1]}) \det_2(y^{[n,n]}) - \det_4(y^{[n,1]})^2}{\det_4(B^{[2,2]})} \tag{27} \]
Figure 11: Computational times in seconds of \texttt{DetPLUinplace}, \( N \)-by-\( d \)-by-\( d \) array with \( N=10^6 \) and \( d \in [2, 10] \) for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.
It is clear that the matrices $M_{r1}, M_{ns}$, $M_{rn,ns}$, and $B_{r2}$ are also symmetric positive definite. Due to symmetry we have $M_{[n,1]} = M_{[1,n]}$ 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 $M_{[n,1]}$.

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

**Algorithm 49** Function $\text{detMixed}$, returns determinant of the symmetric/hermitian positive definite matrix $B$ by using formula (27).

**Input** $B$: a $n$-by-$n$ matrix

**Output** $r$: the scalar $\det B$.

```plaintext
Function $r \leftarrow \text{detMixed}(B)$
if $n = 1$ then
    $r \leftarrow B(1,1)$
else if $n = 2$ then
    $r \leftarrow B(1,1) \ast B(2,2) - B(1,2) \ast B(2,1)$
else
    $r \leftarrow \text{detMixed}(B(2:n,2:n)) \ast \text{detMixed}(B(1:n-1,1:n-1))$
    $r \leftarrow r - \detLaplace(B(1:n-1,2:n)) \ast 2$
    $r \leftarrow r/\text{detMixed}(B(2:n-1,2:n-1))$
end if
end Function
```

A first vectorized and recursive function using the 3D array $A$ is easy to write and it is given in Algorithm 50.

**Algorithm 50** Function $\text{detMixVec}$, returns determinants of symmetric/hermitian positive definite matrices $A_k$ by using formula (27) (vectorized and recursive).

**Input** $A$: $N$-by-$d$-by-$d$ 3D array such that $A(k,:, :) = A_k, \forall k \in [1,N]$.

**Output** $D$: array of size $N$ such that $D(k) = \det(A_k), \forall k \in [1,N]$.

```plaintext
Function $D \leftarrow \text{detMixVec}(A)$
if $d = 1$ then
    $D \leftarrow A(:,1,1)$
else if $d = 2$ then
    $D \leftarrow A(:,1,1) \ast A(:,2,2) - A(:,1,2) \ast A(:,2,1)$
else
    $I_1 \leftarrow 2:d, I_d \leftarrow 1:d-1, I_{1d} \leftarrow 2:d-1$
    $D \leftarrow \text{detMixVec}(A(:,I_1,I_d)) \ast \text{detMixVec}(A(:,I_{1d},I_d))$
    $D \leftarrow D - \detLapVec(A(:,I_1,I_d)) \ast 2$
    $D \leftarrow D/\text{detMixVec}(A(:,I_{1d},I_d))$
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 \(A\) 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 \texttt{DetMix\_Vec\_v04}, returns determinants of the symmetric definite positive matrices \(A_k\) by using formula \texttt{26} (vectorized, recursive and memory safe).

**Input** \(A\) : \(N\)-by-\(d\)-by-\(d\) 3D array such that \(A(k,:,:) = A_k\), \(\forall k \in [1,N]\).
**Input** \(I\) : (optional) row indices, default 1 : \(d\).
Always the same size as \(J\).
**Input** \(J\) : (optional) column indices, default 1 : \(d\).
Always the same size as \(I\).

**Output** \(D\) : array of size \(N\) such that \(D(k) = \text{det}(A_k), \forall k \in [1,N]\).

```matlab
function D = DetMix_vec_v04(A, I, J)
    if I == [] & J == []
        m = d
        I = 1 : d, J = 1 : d
    else
        m = length(I)
    end if
    if m == 1
        D = A(:,1,1)
    elseif m == 2
        D = A(:,1,1).*A(:,2,2) - A(:,1,2).*A(:,2,1)
    else
        I_1 = I(2:m), J_1 = J(2:m),
        I_d = I(1:m-1), J_d = J(1:m-1),
        I_d_1 = I(2:m-1), J_d_1 = J(2:m-1),
        D = DetMix\_Vec\_vec(A, I_1, J_1).*DetMix\_Vec\_vec(A, I_d, J_d)
        D = D - DetLap\_Vec\_vec(A, I_1, I_d), 2
        D = D./DetMix\_Vec\_vec(A, I_d_1, J_d_1)
    end if
end function
```

<table>
<thead>
<tr>
<th>(N)</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>20000</td>
<td>0.013(s)</td>
<td>0.011(s)</td>
<td>0.012(s)</td>
</tr>
<tr>
<td>40000</td>
<td>0.025(s)</td>
<td>0.026(s)</td>
<td>0.024(s)</td>
</tr>
<tr>
<td>60000</td>
<td>0.043(s)</td>
<td>0.033(s)</td>
<td>0.036(s)</td>
</tr>
<tr>
<td>80000</td>
<td>0.061(s)</td>
<td>0.046(s)</td>
<td>0.048(s)</td>
</tr>
<tr>
<td>100000</td>
<td>0.076(s)</td>
<td>0.048(s)</td>
<td>0.060(s)</td>
</tr>
<tr>
<td>500000</td>
<td>0.243(s)</td>
<td>0.239(s)</td>
<td>0.329(s)</td>
</tr>
<tr>
<td>1000000</td>
<td>0.477(s)</td>
<td>0.464(s)</td>
<td>0.650(s)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(N)</th>
<th>Matlab</th>
<th>Octave</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>20000</td>
<td>0.008(s)</td>
<td>0.006(s)</td>
<td>0.009(s)</td>
</tr>
<tr>
<td>40000</td>
<td>0.014(s)</td>
<td>0.014(s)</td>
<td>0.018(s)</td>
</tr>
<tr>
<td>60000</td>
<td>0.025(s)</td>
<td>0.024(s)</td>
<td>0.027(s)</td>
</tr>
<tr>
<td>80000</td>
<td>0.036(s)</td>
<td>0.028(s)</td>
<td>0.036(s)</td>
</tr>
<tr>
<td>100000</td>
<td>0.045(s)</td>
<td>0.032(s)</td>
<td>0.045(s)</td>
</tr>
<tr>
<td>500000</td>
<td>0.160(s)</td>
<td>0.154(s)</td>
<td>0.249(s)</td>
</tr>
<tr>
<td>1000000</td>
<td>0.313(s)</td>
<td>0.295(s)</td>
<td>0.498(s)</td>
</tr>
</tbody>
</table>

(a) Function \texttt{DetMix\_Vec}  (b) Function \texttt{DetMix\_Vec\_vec}

Table 32: Computational times in seconds of \texttt{DetMix} functions with \(A \in (M_{3,3}(\mathbb{K}))^N\) 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:

\[ A \leftarrow B \]  
Assignment

\[ A * B \]  
matrix multiplication,

\[ A .* B \]  
 element-wise multiplication,

\[ A / B \]  
 element-wise division,

\[ A(:) \]  
all the elements of \( A \), regarded as a single column.

\[ A^T \]  
transpose of the matrix \( A \).

\[ [\cdot] \]  
Horizontal concatenation,

\[ [:] \]  
Vertical concatenation,

\[ A(:, J) \]  
\( J \)-th column of \( A \),

\[ A(I,:) \]  
\( I \)-th row of \( A \).

\[ \text{Sum}(A, \text{dim}) \]  
sums along dimension \text{dim},

\[ \text{Prod}(A, \text{dim}) \]  
product along dimension \text{dim},

\[ \text{Size}(A) \]  
return the dimensions of the multi-array \( A \).

\[ \text{Argmax}(A, \text{dim}) \]  
the arguments of the maxima along dimension \text{dim},

\[ \text{Argmin}(A, \text{dim}) \]  
the arguments of the minimum along dimension \text{dim},

\[ I_{m \times n} \text{ (or } I_n \text{)} \]  
n-by-\( n \) identity matrix,

\[ O_{m \times n} \text{ (or } O_n \text{)} \]  
m-by-\( n \) (or \( n \)-by-\( n \)) matrix or sparse matrix of ones,

\[ \text{Ones}(n_1, n_2, \ldots, n_\ell) \]  
\( \ell \)-dimensional array of ones,

\[ \text{Zeros}(n_1, n_2, \ldots, n_\ell) \]  
\( \ell \)-dimensional array of zeros,

\[ \text{RepTile}(A, m, n) \]  
tiles the \( p \)-by-\( q \) array/matrix \( A \) to produce the \((m \times p)\)-by-\((n \times q)\) array composed of copies of \( A \),

\[ \text{Reshape}(A, m, n) \]  
returns the \( m \)-by-\( n \) array/matrix whose elements are taken columnwise from \( A \),

\[ \text{Det}(A) \]  
return the determinant of the square matrix \( A \).

\[ \text{Sub2Ind}(\text{dims}, i_1, i_2, \ldots, i_d) \]  
return the linear index corresponding to the provided subscripts of an array of dimensions \text{dims}. Here \( d \) is the number of dimensions i.e. the length of the \text{dims} array.

\[ \text{Ind2Sub}(\text{dims}, index) \]  
return the subscripts of corresponding to the provided linear index of an array of dimension \text{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-dimensional 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-\( \ldots \)-by-\( n_a \) array and \( B \) be a \( m_1 \)-by-\( m_2 \)-by-\( \ldots \)-by-\( m_b \) with \( n_a \leq m_b \). These two arrays are compatible for element wise operations if

\[ n_i = m_i \text{ or } n_i = 1 \text{ or } m_i = 1, \quad \forall i \in [1, n_a] \]
Let $\circ$ denote an element wise binary operator. If the two arrays $A$ and $B$ are compatible then the following operations are allowed

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

The result arrays $C$ and $D$ have the same dimension $\max(p_1, m_1)$-by-$\max(m_2, m_2)$-by-$\ldots$-by-$\max(n_u, m_u)$-by-$m_{u+1}$-by-$\ldots$-by-$m_b$.

### A.1.1 Sub2ind function

$I \leftarrow \text{Sub2ind}([d_1, \ldots, d_n], i_1, \ldots, i_n)$ returns the linear index $I$ corresponding to the provided subscripts $i_1, \ldots, i_n$ of an $n$-dimensional array of dimensions $[d_1, \ldots, d_n]$. Subscripts $i_1, \ldots, i_n$ must have the same size and index $I$ will have this size. For example if $\mathbf{A}$ is an $n$-dimensional array and all subscripts $i_1, \ldots, i_n$ are 1-dimensional array of dimension $m$ then

$$I \leftarrow \text{Sub2ind}(\text{Size}(\mathbf{A}), i_1, \ldots, i_n)$$

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

$$\mathbf{A}(I(k)) = \mathbf{A}(i_1(k), \ldots, i_n(k)), \quad \forall k \in [1, m]$$

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(V)**

where $V$ is an array of length $n$. Returns a $n!$-by-$n$ array containing all permutations of $V$ elements.

The lexicographical order is chosen.

**Combs(V, k)**

where $V$ is an array of length $n$ and $k \in [1, n]$. 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 developers

git informations on the \LaTeX\ repository of this report

| name: LinAlg3D | tag: | commit: c39bf1b6c6f60e032b90926021e307e0c2932 | 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: 7d9308bda7d0e61906a71d1a16338933bf6 | date: 2018-05-29 | time: 13:26:42 | status: True |
List of algorithms

1. Function $AXpBY_{\text{Cpt}}$, returns linear combination $\alpha X + \beta Y$ by using component by component computation. 
2. Function $\text{getCpt}$, returns component $(i,j)$ of the $k$-th matrix of $X$. 
3. Function $AXpBY_{\text{Cvt}}$, returns linear combination $\alpha X + \beta Y$ by converting arrays to 3D-arrays. 
4. Function $\text{to3Darray}$, converts to a 3D-array. 
5. Function $AXpBY_{\text{Mat}}$, returns linear combination $\alpha X + \beta Y$ by using vectorized operations on 2D-arrays or matrices. 
6. Function $\text{getMat}$, returns the $k$-th matrix of $X$. 
7. Function $AXpBY_{\text{Vec}}$, returns linear combination $\alpha X + \beta Y$ by using vectorized operations on 1D-arrays. 
8. Function $\text{getVec}$, returns $(i,j)$ components of $X$. 

54
Function `by_conv` returns element by element operation \( X \odot Y \). Here \( f \) is the function \( f : (x, y) \in K^2 \rightarrow x \odot y \).

Function `by_mat` returns element by element operation \( X \odot Y \) by using function \( f : (A, B) \rightarrow A \odot B \) where \( A \) and \( B \) are in \( M_{m,n}(K) \) or in \( K \).

Function `by_vec` returns element by element operation \( X \odot Y \) by using function \( f : (A, B) \rightarrow A \odot B \) where \( A \) and \( B \) are in \( K^n \).

Function `by_cvt` returns element by element operation \( X \odot Y \) by converting arrays to a 3D-arrays. Here \( f \) is the function \( f : (A, B) \rightarrow A \odot B \) where \( A \) and \( B \) are in \( (M_{m,n}(K))^2 \).

Function `times_cpt` returns element by element operation \( X \odot Y \) by using vectorized operations on 2D-arrays or matrices.

Function `times_vec` returns element by element operation \( X \odot Y \) by using vectorized operations on 1D-arrays.

Function `times_cvt` returns element by element operation \( X \odot Y \) by converting arrays to a 3D-arrays.

Function `times_cpt` returns matrixial products \( X \odot Y \) where \( X \) or/and \( Y \) are 3D-arrays.

Function `times_vec` returns matrixial products \( X \odot Y \) where \( X \) or/and \( Y \) are 3D-arrays.

Function `times_mat` returns matrixial products \( X \odot Y \) where \( X \) or/and \( Y \) are 3D-arrays.

Function `linsolve_diag` solves diagonal linear system \( AX = B \) where \( A \) is a regular lower triangular matrix.

Function `linsolve_tril_mat` solves equation \( AX = B \) where \( A \) is a regular lower triangular 3D-array(not vectorized).

Function `linsolve_tril_cpt` solves equation \( AX = B \) where \( A \) is a regular lower triangular 3D-array(not vectorized).

Function `linsolve_tril_vec` solves equation \( AX = B \) where \( A \) is a regular lower triangular 3D-array(vectorized).

Function `linsolve_triu` returns solution of equation \( AX = B \) where \( A \) is a regular upper triangular matrix.

Function `linsolve_triu_mat` solves equation \( AX = B \) where \( A \) is a regular upper triangular 3D-array(not vectorized).

Function `linsolve_triu_cpt` solves equation \( AX = B \) where \( A \) is a regular upper triangular 3D-array(not vectorized).

Function `linsolve_triu_vec` solves equation \( AX = B \) where \( A \) is a regular upper triangular 3D-array(vectorized).

Function `cholesky` computes the lower triangular matrix \( L \in M_n(C) \) such that \( B = LL^T \).

Function `cholesky_mat` returns cholesky factorizations of \( A_k \) matrices (not vectorized).

Function `cholesky_cpt` returns cholesky factorizations of \( A_k \) matrices (not vectorized).

Function `cholesky_vec` returns cholesky factorizations of \( A_k \) matrices (vectorized).
33 Function PALU computes the LU factorization with partial pivoting of a matrix A such that PA = LU. 
34 Function pLUinplace inplace computation of the LU factorization with partial pivoting of a matrix A such that PA = LU. 
35 Function PALU_MAT computes all LU factorizations with partial pivoting of a 3D-array A such that PkAk = LkUk. 
36 Function PALU_Cpt computes all LU factorizations with partial pivoting of a 3D-array A such that PkAk = LkUk. 
37 Function PALU_VEC computes all LU factorizations with partial pivoting of a 3D-array A such that PkAk = LkUk. 
38 Function pLUinplace_MAT computes all LU factorizations with partial pivoting of a 3D-array A such that PkAk = LkUk. 
39 Function pLUinplace_Cpt computes all LU factorizations with partial pivoting of a 3D-array A such that PkAk = LkUk. 
40 Function pLUinplace_VEC computes all LU factorizations with partial pivoting of a 3D-array A such that PkAk = LkUk. 
41 Function LinsolveCholesky_VEC solves equation AX = B where A is a symmetric positive definite 3D-array (vectorized). 
42 Function LinsolvePAULU_VEC solves equation AX = B where A is a regular 3D-array (vectorized). 
43 Function Det_MAT returns determinants of a 3D-array (not vectorized). 
44 Function DetLAP returns determinant of the matrix B by using Laplace formula (24) expanded with respect to the 1-st row. 
45 Function DetLAP_MAT returns determinants of a 3D-array (not vectorized). 
46 Function DetLAP_Cpt returns determinants of a 3D-array in (M_{d,d}(K))^n (not vectorized). 
47 Function DetLAP_VEC returns determinants of a 3D-array in (M_{d,d}(K))^n (not vectorized). 
48 Function DetLAPmix returns determinants of A_k matrices by using Laplace formula (24) expanded with respect to the 1-st row (vectorized, recursive and memory safe). 
49 Function DetMixed returns determinant of the symmetric/hermitian positive definite matrix B by using formula (27). 
50 Function DetMix_VEC returns determinants of symmetric/hermitian positive definite matrices A_k by using formula (27) (vectorized and recursive). 
51 Function DetVec_v04 returns determinants of the symmetric definite positive matrices A_k by using formula (26) (vectorized, recursive and memory safe). 

List of Tables

1. Common element by element operations 
2. Computational times in seconds of AXPY functions with X ∈ (M_{d,d}(K))^n and Y ∈ M_{d,d}(R) for Matlab 2018a, Octave 4.4.0 and Python 3.6.5.
| Computational times in seconds of $\text{AXpbY}_{\text{MAT}}$ functions with |
| $X$ in $(M_{3,3}(K))^N$ and $Y$ in $M_{3,3}(R)$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. Matlab(*) refers to Matlab without multi-threadings. | 11 |

| Computational times in seconds of $\text{AXpbY}_{\text{VEC}}$ functions with |
| $X$ and $Y$ in $(M_{3,3}(K))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. Matlab(*) refers to Matlab without multi-threadings. | 11 |

| Computational times in seconds of $\text{AXpbY}_{\text{CVT}}$ functions with |
| $X$ in $(M_{3,3}(K))^N$ and $Y$ in $M_{3,3}(R)$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. Matlab(*) refers to Matlab without multi-threadings. | 12 |

| Computational times in seconds of $\text{AXpbY}_{\text{MAT}}$ functions with |
| $X$ in $(M_{3,3}(K))^N$ and $Y$ in $M_{3,3}(R)$ under Matlab 2018a: effects of multithreading on cputimes | 12 |

| Computational times in seconds of $\text{AXpbY}_{\text{VEC}}$ functions with |
| $X$ in $(M_{3,3}(K))^N$ and $Y$ in $M_{3,3}(R)$ under Matlab 2018a: effects of multithreading on cputimes | 12 |

| Computational times in seconds of $\text{AXpbY}_{\text{CVT}}$ functions with |
| $X$ in $(M_{3,3}(K))^N$ and $Y$ in $M_{3,3}(R)$ under Matlab 2018a: effects of multithreading on cputimes | 12 |

| Computational times in seconds of the |
| $\text{mtimes}_{\text{Vec}}$ functions with $X$ in $(M_{3,3}(K))^N$ and $Y$ in $M_{3,3}(R)$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. | 13 |

| Computational times in seconds of the |
| $\text{mtimes}$ functions with $X$ in $(M_{3,3}(K))^N$ and $Y$ in $M_{3,3}(R)$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. | 15 |

| Computational times in seconds of the |
| $\text{mtimes}_{\text{Vec}}$ functions with $X$ in $(M_{3,3}(K))^N$ and $Y$ in $M_{3,3}(R)$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. | 15 |

| Computational times in seconds of the |
| $\text{LinsolveTriU}_{\text{Vec}}$ functions with $\text{A} \in (M_{3,3}(K))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. | 20 |

| Computational times in seconds of the |
| $\text{LinsolveTriU}_{\text{Vec}}$ functions with $\text{A} \in (M_{3,3}(K))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. | 20 |

| Computational times in seconds of the |
| $\text{LinsolveTriU}$ functions with $\text{A} \in (M_{3,3}(K))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. | 23 |

| Computational times in seconds of the |
| $\text{LinsolveTriU}_{\text{Vec}}$ functions with $\text{A} \in (M_{3,3}(K))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. | 23 |

| Computational times in seconds of the |
| $\text{Cholesky}_{\text{Vec}}$ functions with $\text{A} \in (M_{3,3}(K))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. | 29 |

| Computational times in seconds of the |
| $\text{Cholesky}_{\text{Vec}}$ functions with $\text{A} \in (M_{3,3}(K))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. | 29 |

| Computational times in seconds of the |
| $\text{Cholesky}$ functions with $\text{A} \in (M_{3,3}(K))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. | 29 |

| Computational times in seconds of the |
| $\text{Cholesky}_{\text{Vec}}$ functions with $\text{A} \in (M_{3,3}(K))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. | 29 |

| Computational times in seconds of the |
| $\text{Cholesky}$ functions with $\text{A} \in (M_{3,3}(K))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. | 29 |

| Computational times in seconds of the |
| $\text{PALU}$ functions with $\text{A} \in (M_{3,3}(K))^N$ for Matlab 2018a, Octave 4.4.0 and Python 3.6.5. | 35 |
References

