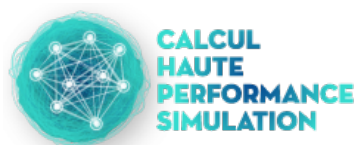




université PARIS-SACLAY



UNIVERSITÉ DE VERSAILLES  
SAINT-QUENTIN-EN-YVELINES  
DÉPARTEMENT DE INFORMATIQUE

MASTER 1 CALCUL HAUTE PERFORMANCE, SIMULATION

---

***TD/TP - Calcul Numérique***

---

*Réalisé par :*

BOUCHELGA ABDELJALIL

*Encadré par :*

Pr. THOMAS DUFAUD

Année Universitaire :2021-2022

# PARTIE 1

## INTRODUCTION

### Introduction

L'objectif de TD/TP 2 est de prendre les automatismes sur la rédaction des algorithmes et leur analyse. Pour cela nous allons écrire des algorithmes numérique et évaluer leur complexité arithmétique et en terme de stockage mémoire et puis comparer leurs performances afin d'implémenter des algorithmes efficace en terme de performance.

Pour le TD/TP 3 l'objectif est de savoir différentes type d'algorithme de résolution d'un système linéaire, ainsi comprendre les performances et les limites de chaque algorithme et puis comparer les performances.

### Outil utilisé

1. Scilab
2. Latex

## PARTIE 2

## TD/TP 2 CALCUL NUMÉRIQUE

### 2.1 Exercice 1 TP

Le but de cet exercice est de se familiariser avec le langage Scilab et savoir les notion basic qu'on aura besoin par la suite.

1. Écrivez un vecteur  $x$  à 1 ligne et 4 colonnes.

```
-->x=[1,2,3,4]
```

x =

1.    2.    3.    4.

2. Écrivez un vecteur  $y$  à 4 lignes et 1 colonnes

```
-->y=[1;2;3;4]
```

y =

1.

2.

3.

4.

3. les opérations

```
-->x=[1;2;3;4]
```

x =

1.

2.

3.

4.

```
-->z=x+y
```

```
z =
```

```
2.  
4.  
6.  
8.
```

```
-->x=[1,2,3,4]
```

```
x =
```

```
1.    2.    3.    4.
```

```
-->s=x*y
```

```
s =
```

```
30.
```

4. la fonction size()

```
-->size(x)
```

```
ans =
```

```
1.    4.
```

```
-->size(y)
```

```
ans =
```

```
4.    1.
```

5. la norme 2 de x avec la fonction norme

```
-->norm(x)
```

```
ans =
```

```
5.4772256
```

6. matrice A à 4 lignes et 3 colonnes

```
-->A=[1,2,3;4,5,6;7,8,9;10,11,12]
```

```
A =
```

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

7. la transposée de A.

```
-->A'
```

```
ans =
```

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

8. les opérations de bases avec deux matrices carrées A et B

```
-->B=[1,1,1;2,2,2;3,3,3;4,4,4]
```

```
B =
```

```
1.  1.  1.
2.  2.  2.
3.  3.  3.
4.  4.  4.
```

```
-->A
```

```
A =
```

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

La somme:

```
-->c=A+B
```

```
c =
```

```
2.  3.  4.
6.  7.  8.
10. 11. 12.
14. 15. 16.
```

Le produit:

```
-->D=A*B'
```

```
D =
```

```
6.  12. 18. 24.
15. 30. 45. 60.
24. 48. 72. 96.
33. 66. 99. 132.
```

La soustraction:

```
-->E=A-B
```

```
E =
```

```
0.    1.    2.
```

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

```
4.    5.    6.
```

```
6.    7.    8.
```

9. le conditionnement de A avec la fonction `cond()`.

```
-->cond(A)
```

```
ans =
```

```
9.882D+15
```

## 2.2 Exercice 2 TP : Matrice random et problème "jouet"

1. Génération d'une matrice A de taille  $3 \times 3$  en utilisant la fonction `rand()`.

```
-->A=rand(3,3)
```

```
A =
```

```
0.2113249    0.3303271    0.8497452
```

```
0.7560439    0.6653811    0.685731
```

```
0.0002211    0.6283918    0.8782165
```

2. Génération d'un vecteur  $xex \in \mathbb{R}$  avec la fonction `rand()`.

```
-->xex=rand(1:3)
```

```
xex =
```

```
0.068374    0.5608486    0.6623569
```

```
-->xex'
```

```
ans =
```

```
0.7263507
```

```
0.1985144
```

```
0.5442573
```

3. Calcul de  $b = A * xex$

```
-->b=A*xex
```

```
b =
```

```
0.6815507
```

```
1.0544548
0.6028812
```

4. Résolution du système  $Ax = b$  avec la fonction " $\backslash$ ".

```
-->x=A\b
x =
```

```
0.2320748
0.2312237
0.2164633
```

**Remarque :** Lorsque  $A$  est carré et non singulière,  $x = A \backslash b$  est équivalent à  $x = \text{inv}(A) * b$  en arithmétique exacte, mais le calcul réalisé par *backslash* est plus précis et et moins coûteux en arithmétique flottante. Par conséquent, pour calculer la solution du système d'équations linéaires  $Ax = b$ , on devrait plutôt utiliser l'opérateur *backslash*, et éviter la fonction *inv* qui numériquement moins stable.

5. Calcul de l'erreur avant et arrière

Erreur avant :

```
-->err_avant=norm(xex-x)/norm(xex)
err_avant =
```

```
3.999D-16
```

Erreur arrière :

```
-->r=b-A*x
r =
```

```
0.
5.551D-17
5.551D-17
```

```
-->relres = norm(r)/norm(A)*norm(x)
relres =
```

```
1.734D-17
```

Conditionnement :

```
-->cond(A)
ans =
```

```
8.2596760
```

6. Test avec différentes tailles :

Pour refaire les 5 points précédentes j'ai créé le script en dessous et à chaque fois je change la taille.

```
function a=jouet(n)
A=rand(n,n)    //Matrice de taille n
c=cond(A)
disp("c =", c)
xex=rand(1:n)  //Vecteur de taille n
xex=xex'
b=A*xex        //calcul de b
x=A\b          //Resolution du systeme
disp("x = ",x)
err_avant=norm(xex-x)/norm(xex) //Erreur Avant
disp("Erreur_Avant = ",err_avant)
r=b-A*x        //Erreur Arriere
disp("r =",r)
relres = norm(r)/(norm(A)*norm(x))
disp("Relres = ",relres)
a=1
endfunction
```

### Analyse des resultats

Nous remarquons que le conditionnement augmente lorsqu'on augmente la taille de la matrice, ce qui justifie l'augmentation des erreurs arrière et avant, ainsi l'erreur avant est toujours supérieure à l'erreur arrière.

## 2.3 Exercice 2 TP : Produit Matrice-Matrice

le but de ce tp est de comparer la complexité de 3 algorithmes qui font le produit de deux matrices.

L'algorithme du produit Matrice-Matrice "ijk" **matmat3b**

```
function [C]= matmat3b(A,B)
m=size(A,1);
p=size(B,1);
n=size(B,2);
C=zeros(m,n);
for i = 1 : m
    for j = 1 : n
        for k = 1 : p
            C(i, j) = A(i, k)*B(k, j) + C(i, j);
        end
    end
end
endfunction
```



L'algorithme du produit Matrice-Matrice "ijk" **matmat2b**

```
function [C]= matmat2b(A,B)
m=size(A,1);
n=size(B,2);
C=zeros(m,n);
for i = 1 : m
    for j = 1 : n
        C(i, j) = A(i, :)*B(:, j) + C(i, j);
    end
end
endfunction
```

L'algorithme du produit Matrice-Matrice "ijk" **matmat1b**

```
function [C]= matmat1b(A,B)
m=size(A,1);
n=size(B,2);
C=zeros(m,n);
for i = 1 : m
    C(i, :) = A(i, :)*B + C(i, :);
end
endfunction
```

### Mesure de temps d'exécution des algorithmes

Pour les mesures de temps des algorithmes, j'ai réalisé 10 itération de mesure pour chaque algorithme, et puis j'ai calculé la moyenne.

Pour la taille n=3 :

<b>matmat1b</b>	<b>matmat2b</b>	<b>matmat3b</b>
0.000182	0.0002840	0.000305
0.0001690	0.000273	0.000326
0.000211	0.0002800	0.000214
0.000215	0.000255	0.000372
0.000176	0.0003340	0.000328
0.000158	0.00023	0.0002810
0.000241	0.000252	0.000297
0.000158	0.000235	0.000335
0.000235	0.000241	0.000366
0.000142	0.000271	0.000312
<b>moy= 0,0001887</b>	<b>moy= 0,000272571</b>	<b>moy= 0,000303286</b>

Pour la taille n=50 :

matmat1b	matmat2b	matmat3b
0.002848	0.022936	0.329394
0.002065	0.021612	0.3223
0.0047470	0.016667	0.32138
0.001994	0.014888	0.3252400
0.001691	0.015411	0.3142060
0.00231	0.0194920	0.314723
0.00241	0.017045	0.335005
0.001952	0.018042	0.32015
0.00235	0.018413	0.33100
0.00192	0.025421	0.325045
<b>moy= 0,002515286</b>	<b>moy= 0,018293</b>	<b>moy= 0,323178286</b>

Pour la taille n=100 :

matmat1b	matmat2b	matmat3b
0.006457	0.072851	2.639783
0.007966	0.070574	2.569376
0.007995	0.072893	2.624991
0.00642	0.0657150	2.591531
0.008452	0.073936	2.65817
0.006965	0.068855	2.71989
0.007161	0.076676	2.603488
0.006965	0.061672	2.61280
0.007961	0.071686	2.71087
0.006988	0.062636	2.71900
<b>moy= 0,007345143</b>	<b>moy= 0,070214286</b>	<b>moy= 2,629604143</b>

## Graphe

Pour bien visualiser les resultats, j'ai relisé le graphe suivant :

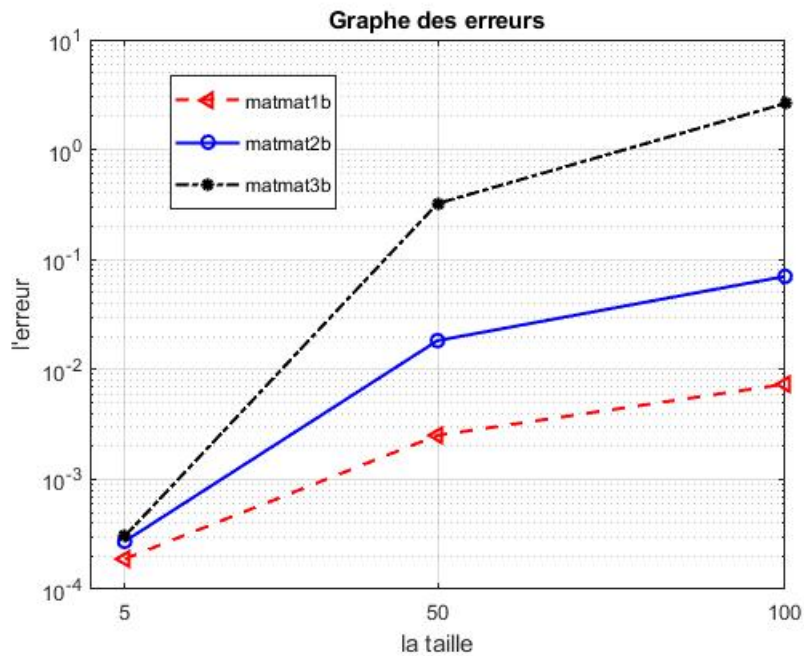
Script pour le graphe Graphe :

```
x=[5,50,100]
B1=[0.0001887,0.002515286,0.007345143];
B2=[0.000272571,0.018293,0.070214286];
B3=[0.000303286,0.323178286,2.629604143];

semilogy(x,B1,'r--<',x,B2,'b-o',x,B3,'k-.*')

hleg1 = legend( ['matmat1b','matmat2b','matmat13b']);
set(hleg1, 'Position', [.7,.71,.1,.2]);
xlabel('la taille');
```

```
ylabel('l'erreur');
title('Graphe des erreurs')
```



### Analyse des resultats

les trois algorithmes ont une une complexité differentes, le produit matrice-matrice de l'algorithme matmat3b qui déroule 3 boucle à une complexité cubique et la l'agorithme matmat2b à une complexité quadratique.

Les mesures de temps que nous avons fait pour les 3 algorithmes et pour les differentes taille, nous montre toujours que l'algorithme matmat1b qui a une complexité lineaire est le plus rapide.

## PARTIE 3

### TD/TP 3 CALCUL NUMÉRIQUE RÉOLUTION DE SYSTÈME LINÉAIRE

#### 3.1 Exercice 1 TP Système Triangulaire

Les méthodes directes reposent sur une décomposition de la matrice  $A$  en  $A = LU$  où  $L$  est facile à inverser (triangulaire ou orthonormale) et  $U$  est triangulaire. On aura donc

$$Ax = b \Leftrightarrow LUx = b \Leftrightarrow \begin{cases} Lx = b \\ Ux = y \end{cases}$$

Il faut donc savoir résoudre le système  $Ax = b$  lorsque  $A$  est triangulaire inférieure (ou supérieure).

Dans cet exercice nous allons voir les deux méthodes de résolution des systèmes triangulaires, la méthode de remontée et de descente.

1. L'algorithme de la résolution par remontée : *usolve*

```
function [x]= usolve(U,b)
    n = size(U,1);
    x = zeros(n);
    x(n) = b(n)/U(n, n);
    for i = n-1:-1:1
        x(i) = (b(i) - U(i, (i+1):n)*x((i+1):n))/U(i,i);
    end
endfunction
```

2. L'algorithme de la résolution par descente : *lsolve*

```
function [x]= lsolve(L,b)
    n = size(L,2);
    x = zeros(n);
    x(1) = b(1)/L(1, 1);
```

```

for i = 2:n
x(i) = (b(i) - L(i, 1:(i - 1))*x(1:(i - 1)))/L(i, i);
end
endfunction

```

### 3. Test et validation des algorithmes

Nous allons utiliser les deux fonctions **tril()** et **triu()** qui permettant d'extraire une matrice triangulaire inférieur ou supérieur.

**usolve :**

```

-->A
A =

0.7560439    0.6653811    0.685731
0.0002211    0.6283918    0.8782165
0.3303271    0.8497452    0.068374

-->U=tril(A)
U =

0.7560439    0.        0.
0.0002211    0.6283918    0.
0.3303271    0.8497452    0.068374

-->b=A*xex
b =

1.3377668
1.0127856
0.5407226

-->usolve(U,b)
ans =

1.7694302
1.6117104
7.9083027

lsolve :

-->L=triu(A)
L =

0.7560439    0.6653811    0.685731
0.          0.6283918    0.8782165
0.          0.          0.068374

```

```
-->lsolve(L,b)
ans =
```

```
1.7694302
1.6117104
7.9083027
```

## 3.2 Exercice 2 TP Gauss

Le principe de la méthode de Gauss est de se ramener, par des opérations simples, à un système triangulaire équivalent, qui sera donc facile à inverser.

1. l'algorithme de résolution par élimination de Gauss sans pivotage.

```
A=rand(3,3)
b=rand(1:3)
b=b'
function [x]= gausskij3b(A,b)
    n = size(A,1);
    x = zeros(n,1);
    for k = 1 : n - 1
        for i = k + 1 : n
            mik = A(i, k)/A(k, k);
            b(i) = b(i) - mik * b(k);
            for j = k + 1 : n
                A(i, j) = A(i, j) - mik * A(k, j);
            end
        end
    end
    x=usolve(A,b);
endfunction
```

2. test et validation

Nous allons résoudre le système  $Ax = b$  en utilisant la fonction *Backslash* et la méthode de gauss.

```
-->A=rand(3,3)
A =
```

```
0.7560439    0.6653811    0.685731
0.0002211    0.6283918    0.8782165
0.3303271    0.8497452    0.068374
```

```
-->b=rand(1:3)'
```

```

b  =

0.9329616
0.2146008
0.312642

-->x=A\b
x  =

-0.7943407
0.6160788
2.3259083

-->gausskij3b(A,b)
ans =

-0.7943407
0.6160788
2.3259083

```

Les deux methodes donnent les memes resultats.

### 3.3 Exercice 3 TP LU

L'idée de la factorisation de la matrice  $A$ , est l'écrire comme un produit  $A = LU$ , où  $L$  est triangulaire inférieure et  $U$  triangulaire supérieure et on reformule alors le système  $Ax = b$  sous la forme  $LUx = b$  et on résout maintenant deux systèmes faciles à résoudre car triangulaires :  $Ly = b$  et  $Ux = y$ . La factorisation  $LU$  de la matrice découle immédiatement de l'algorithme de Gauss.

#### 1. l'algorithme de factorisation LU

```

function [L,U]=mylu3b(A)
n = size(A,1);
L=zeros(n,n);
U=zeros(n,n);
for k = 1 : n - 1
    for i = k + 1 : n
        A(i, k) = A(i, k)/A(k, k);
    end
    for i = k + 1 : n
        for j = k + 1 : n
            A(i, j) = A(i, j) - A(i, k) * A(k, j);
        end
    end
end

```

```

end
L = tril(A);
U = triu(A);
endfunction

```

## 2. Test et validation

```
-->A=rand(3,3)
```

```
A =
```

```

0.7560439    0.6653811    0.685731
0.0002211    0.6283918    0.8782165
0.3303271    0.8497452    0.068374

```

```
-->[l,u]=lu(A)
```

```
l =
```

```

1.          0.          0.
0.0002925    1.          0.
0.4369153    0.8898959    1.

```

```
u =
```

```

0.7560439    0.6653811    0.685731
0.          0.6281972    0.8780159
0.          0.          -1.0125751

```

```
-->l*u
```

```
ans =
```

```

0.7560439    0.6653811    0.685731
0.0002211    0.6283918    0.8782165
0.3303271    0.8497452    0.068374

```

Calcul de l'erreur commise sur la factorisation  $LU : A - LU$

```
-->Er=A-LU
```

```
Er =
```

```

0.    0.    0.
0.    0.    0.
0.    0.    0.

```

## 4. La methode de pivot partiel

```

function [P,L,U]=mylu(A)
n=size(A,1)
P=zeros(n,n)
P(n,n)=1
for k=1:n-1

```



```

pivot_max=max(abs(A(k:n,k)))
P(k,find(pivot_max==A(k:n,k),1))=1
A([k find(pivot_max==A(k:n,k),1)],:)=A([find(pivot_max==A(k:n,k),1) k],: )
for i=k+1:n
    A(i,k)=A(i,k)/A(k,k)
    for j=k+1:n
        A(i,j)=A(i,j)-(A(i,k)/A(k,k))*A(k,j)
    end
end
end
U=triu(A)
L=tril(A)
for l=1:n
    L(l,l)=1
endfunction

```