Rapport Master M1

Aicha Maaoui

Master Calcul Haute Performance Simulation (CHPS)

TP2-TP3 Calcul Numerique

Novembre 2021

Institut des Sciences et Techniques des Yvelines (ISTY)

Abstract

Les TPs 2 et 3 "Résolution de Systèmes Linéaires" sont associés au cours de Calcul Numérique sur l'algèbre linéaire.

Ils ont pour but:

- Ecrire des algorithmes numériques
- Evaluer la complexité arithmétique et en espace
- Implémenter un algorithme pour le calcul numérique (Scilab)
- Evaluer la complexité d'un algorithme
- Comprendre les performances et limites d'un algorithme
- Comparer les performances
- Résoudre des systèmes linéaires.

Contents

List of Figures

List of Tables

Listings

Chapter 1

TP2 de Calcul Numérique

Les exercices 6, 7 et 8 du TP2 sont réalisés sur scilab. Ce compte rendu comprend l'analyse des algorithmes à implémenter, les observations et analyses des algorithmes, ainsi que des mesures de performances.

1.1 Exercice 6: Prise en main sur Scilab

1/ Vecteur $x \in R^{1\times 4}$ (1 ligne, 4 colonnes): $x = [2, 4, 6, 8]$.

2/ Vecteur $y \in R^{4\times 1}$ (4 ligne, 1 colonnes): $y = [7; 1; 8; 2]$.

3/ **Addition:** On ne peut pas additionner un vecteur $x \in R^{1\times 4}$ et un vecteur $y \in R^{4\times 1}$. Par conséquent, on utilise la transposition du vecteur y (on peut aussi faire la transposition du vecteur x et le sommer avec le vecteur y). Soit le vecteur z le résultat de la sommation des vecteurs $x \in R^{1\times 4}$ et $y^T \in R^{1\times 4}$. Alors: $z = x + y' \Rightarrow z = [9, 5, 14, 10]$.

Multiplication: On peut multiplier un vecteur $x \in R^{1\times 4}$ et un vecteur $y \in R^{4\times 1}$ (nombre de colonnes du premier vecteur = nombre de lignes du deuxième vecteur). Soit le scalaire s le résultat de la multiplication des vecteurs x et y. Alors: $s = x \times y \Rightarrow s = 82$.

4/ **Calcul de taille des vecteurs x et y:** Pour calculer les tailles des vecteurs x et y, on utilise size(x) et size(y). On obtient alors dans Scilab: $sizeX = 1. \quad 4.$ et $sizeY = 4. \quad 1.$

5/ **Norme 2 de x:** Pour calculer la norme 2 du vecteur x, on introduit dans Scilab la commande: norm(x,2) Ainsi, le résultat obtenu est évalué à: 10.954451.

6/ **Matrice $A \in R^{4\times 3}$ (4 lignes et 3 colonnes):** Pour génerer une matrice A à 4 lignes et 3 colonnes, on introduit dans Scilab la commande: A= [1, 3, 7, 13; 1, 5, 9, 71; 8, 10, 50, 45]

(on peut utiliser la commande $A = rand(4, 3)$ pour génerer une matrice aléatoire à 4 lignes et 3 colonnes dans Scilab).

Le résultat de la matrice A introduite est:

$A =$

1. 3. 7. 13.

1. 5. 9. 71.

8. 10. 50. 45.

7/ **Transposée de la matrice A:** Pour calculer la transposée A^T de la matrice A, on utilise la commande $TransA = A'$ de Scilab.

Ainsi, la matrice transposée A^T s'écrit dans Scilab:

$TransA =$

1. 1. 8.

3. 5. 10.

7. 9. 50.

13. 71. 45.

La matrice transposée de A, $TransA$ est une matrice à 3 lignes et 4 colonnes: $TransA \in R^{3\times4}$.

8/ **Opérations de bases avec deux matrices carrées A et B:**

Soit A et B deux matrices carrées $A, B \in R^{4\times4}$.

On introduit dans Scilab les deux matrices carrées A et B à 4 lignes et 4 colonnes:

$A = [1,\ 3,\ 7,\ 13;\ 1,\ 5,\ 9,\ 71;\ 8,\ 10,\ 50,\ 45;\ 8,\ 7,\ 5,\ 1]$

$B = [18,\ 0,\ 1,\ 13;\ 21,\ 45,\ 19,\ 0;\ 18,\ 20,\ 51,\ 45;\ 3,\ 7,\ 9,\ 26]$

Puis, on effectue les operations suivantes:

□ Addition de deux matrices A et B:

On note sum la matrice carrée $\in R^{4\times4}$, somme de deux matrices A et B: $sum = A + B$

Ainsi, le résultat de l'addition de A et B, stocké dans la matrice sum:

$sum =$

19. 3. 8. 26.

22. 50. 28. 71.

26. 30. 101. 90.

11. 14. 14. 27.

☐ Multiplication de deux matrices A et B:

On note mul la matrice carrée $\in R^{4\times4}$, produit de deux matrices A et B: $mul = A \times B$

Ainsi, le résultat de la multiplication de A et B, stocké dans la matrice mul:

$mul =$

246.	366.	532.	666.
498.	902.	1194.	2264.
1389	1765	3153.	3524.
384.	422.	405.	355.

☐ Matrices Transposées de A et B:

On note $C1$ et $C2$ les matrices transposées de A et $B \in R^{4\times4}$.

Dans Scilab: $C1 = A'$ et $C2 = B'$

☐ Multiplication deux matrices A et B par un scalaire $lambda$:

On pose le scalaire: $lambda = 50$

Soit matrices $A1$ et $B1$ les matrices $\in R^{4\times4}$, obtenues en multipliant les matrices A et B par un scalaire, respectivement.

Dans Scilab: $A1 = lambda \times A$ et $B1 = lambda \times B$

9/ **Conditionnement de la matrice A:** Le conditionnement de la matrice A est calculé dans Scilab à l'aide de la fonction $cond(A)$. Ainsi, il est évalué à 95.589276, qui est supérieur à 1.

En effet, on note $K(A)$ le conditionnement de la matrice A. Alors:

$$\begin{cases} K(A) = ||A|| \times ||A^{-1}||; \quad A \in R^{n\times n} \\ ||I|| = 1 \ \ (Matrice\ identité) \\ \Rightarrow I = A \times A^{-1} \Rightarrow ||I|| = ||A|| \times ||A^{-1}|| \end{cases}$$

Ainsi, $||A \times A^{-1}|| \leq ||A|| \times ||A^{-1}|| = K(A) \Rightarrow K(A) \geq 1$.

1.2 Exercice 7: Matrice random et problème "Jouet"

1/ **Matrice Random $A \in R^{3\times3}$:** Pour générer une matrice A aléatoire à l'aide de Scilab, on utilise la commande $A = rand(3, 3)$.

La matrice aléatoire A générée est alors:

$A =$

0.5819606 0.6603468 0.8009798

0.9860136 0.5958275 0.5262907

0.249952 0.5903676 0.1496625

2/ **Vecteur Random** $xex \in R^{3\times1}$**:** Pour générer un vecteur colonne xex aléatoire à l'aide de Scilab, on utilise la commande $xex = rand(3, 1)$.

Le vecteur aléatoire xex généré dans Scilab est alors:

$xex =$

0.0642968

0.4806157

0.6156906

→ xex *est un vecteur colonne.*

3/ **Produit du matrice** A **et vecteur** xex**:** Soit b le produit du matrice A et du vecteur xex. Alors: $b = A \times xex$. Ainsi, le vecteur $b \in R^{3\times1}$ obtenu est:

0.847947

0.6737939

0.3919569

4/ **Résolution du système** $A \times x = b$**:** Le vecteur x est calculé dans Scilab à l'aide de la commande $x = (A\backslash b)$. Alors x est égal:

0.0642968

0.4806157

0.6156906

5/ **Erreur Avant et Arrière:** On utilise la norme 2 dans le calcul des erreurs avant et arrière, qui est aussi la norme par défaut dans Scilab.

- Calcul de l'erreur avant: $||\frac{xex-x}{xex}||$

  On utilise la commande $err = norm((xex - x)/xex, 2)$ dans Scilab

- Calcul de l'erreur arrière: $\frac{||b-A\times x||}{||A||\times||x||}$

  On utilise la commande $relres = norm(b-A*x, 2)/(norm(A, 2)*norm(x, 2))$ dans Scilab.

Alors, les erreurs calculées avec Scilab sont les suivantes:

$$\begin{cases} err = 9.016e-17 \\ relres = 7.894e-17 \end{cases}$$

Erreurs de computation:

L'erreur en arrière est une mesure de combien les données doivent être perturbées pour produire la solution approximative associée à un problème (aussi utile pour la caractérisation des erreurs de troncature).

L'erreur avant est la différence entre les solutions approximée et vraie. Elle est encadrée par (Oliveira 2021, *"What Is Backward Error?"* 2020):

$$Erreur\ avant \preceq Nombre\ de\ conditionnement \times Erreur\ arrière$$

En effet, dans notre cas:

$$\begin{cases} cond(A) = 5.6459931 > 1 \\ Erreur\ avant = 9.016e-17 \\ Erreur\ arrière \times cond(A) = 4.457e-16 \end{cases}$$

Donc l'inégalité précédente est approximativement vérifiée.

6/ **Erreur Avant et Arrière en variant la tailles de matrice A:**

Les points de 1 à 5 sont repétés pour différentes tailles de matrice A n $\in \{100, 300, 500, 800, 900, 1000\}$.

Les variations de ces erreus en fonction de n sont regroupés dans le tableau 1.

Taille du Matrice n	**Erreur Avant**	**Erreur Arrière**
100	4.25e-13	3.158e-16
300	4.20e-12	6.48e-16
500	1.08e-11	9.60e-16
800	1.51e-11	1.20e-15
900	5.70e-11	1.32e-15
1000	1.66e-10	1.08e-15

Table 1.1: Variation des erreurs avant et arrière en fonction de la taille de matrice A.

Les courbes illustrants la variation des erreurs avant et arrière en fonction de la taille de matrice n sont tracé en utilisant Scilab, comme indiqué sur la figure 1.1.

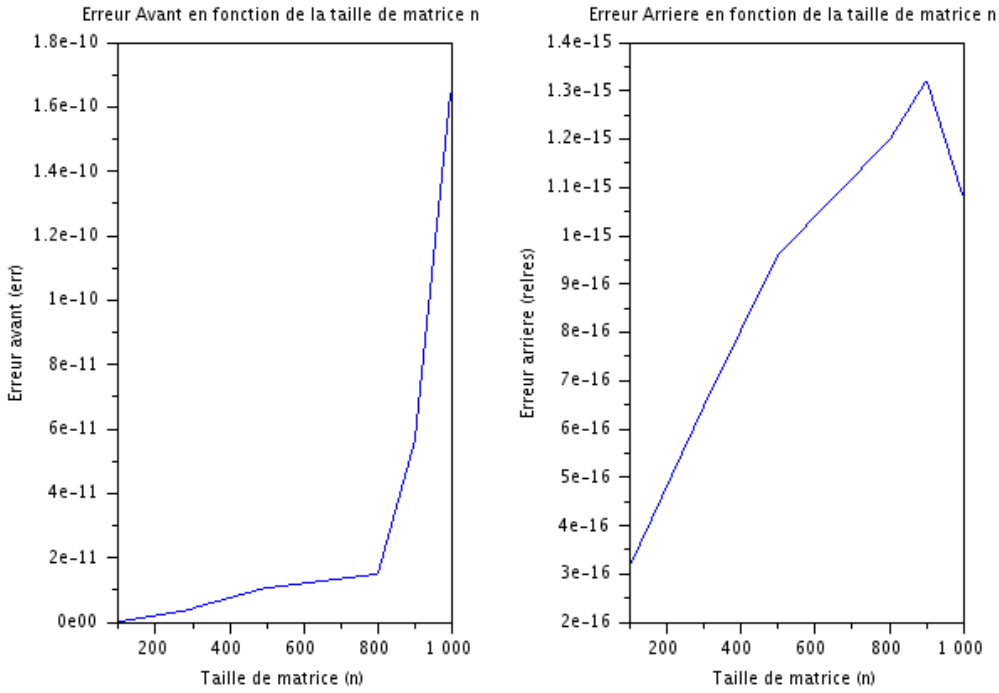

Figure 1.1: Variation de l'erreur avant et arrière en fonction de la taille de matrice A.

D'apres la figure 1.1, l'erreur arrière augmente pour un certain nombre de taille n. Elle représente dans ce cas les erreurs d'arrondi. Pour $n \approx 900$, l'erreur diminue. On a dans ce cas les erreurs de troncature. D'autre part, l'erreur avant augmente en fonction de n.
Pour la taille de la matrice A, $n = 10000$, la génération d'une matrice aléatoire A des valeurs double doit stocker $n^2 = 10000 \times 10000 = 10^8$ cases mémoires, c'est à dire 8×10^8 octets (64bits).

Donc, la génération d'une matrice A carée de taille 10000×10000 consomme énormément en mémoire et en temps. On peut estimer le temps de la CPU pour générer la matrice aléatoire A de la manière suivante:

Listing 1.1: Calcul du Temps écoulé

```
n = 10000; %Taille de la matrice A
A = rand(n,n); xex=rand(n,1);

timer ();
tic ();
%b=A*xex;
A = rand(n,n); %output: 5.719562   5.961835   1.0423587
tUser = toc(); %Temps User
tCpu = timer (); %Temps CPU
disp([ tUser tCpu tCpu/tUser])
```

1.3 Exercice 8: Produit Matrice-Matrice

Dans ce qui suit, les fonctions produit Matrice-Matrice sont inclus dans des SciNotes. Chaque fonction est ensuite testée avec des valeurs aléatoires de matrice A et B. Le temps de calcul est mesuré avec les fonctions tic() et toc() de Scilab.

La compléxité des algorithmes est ensuite étudiée en terme d'opérations éffectuées.

1/ **Algorithme du produit Matrice-Matrice "ijk" à 3 boucles:**

Le produit Matrice-Matrice "ijk" à 3 boucles implémenté dans Scilab est donné dans listing 1.2.

Le produit du matrice A et matrice B est stocké dans la matrice C. En effet,

$$\forall i, j : c_{ij} = c_{ij} + \sum_{k} a_{ik} \times b_{kj}$$

Listing 1.2: Produit Matrice-Matrice "ijk" avec 3 boucles

```
function[C]=matmat3b(A, B)

%Fonction produit Matrice-Matrice "ijk" a 3 boucles

[m p] = size(A) %Taille de la matrice A
[p n] = size(B) %Taille de la matrice B
C=zeros(m,n);   %Initiation de la matrice C

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
funcprot(0)
```

Etude de Complexité:

On se base sur le nombre d'opérations dans le calcul de la complexité, comme suit:

- On a $\sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{p} 1 = p \times n \times m$ multiplications,
- On a $\sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{p} 1 = p \times n \times m$ additions.

Au total, on a $(2 \times p \times n \times m)$ opérations.

La complexité de l'algorithme Produit Matrice-Matrice "ijk" avec 3 boucles est: o$(2m \times p \times n)$ et O$(m \times p \times n)$.

2/ **Algorithme du produit Matrice-Matrice "ijk" à 2 boucles:**

Le produit Matrice-Matrice "ijk" à 2 boucles implémenté dans Scilab est donné dans listing 1.3.

Listing 1.3: Produit Matrice-Matrice "ijk" avec 2 boucles

```
function[C]=matmat2b(A, B)

%Fonction produit Matrice-Matrice "ijk" a 2 boucles

function[C]=matmat2b(A, B)
[m p] = size(A) %Taille de la matrice A
[p n] = size(B) %Taille de la matrice B
C=zeros(m,n);   %Initiation de la matrice C

for i = 1:m
    for j = 1:n
            C(i,j)=A(i,:)*B(:,j)+C(i,j);
    end
end

endfunction
funcprot(0)
```

Etude de Complexité:

On se base sur le nombre d'opérations dans le calcul de la complexité, comme suit:

- On a $\sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{p} 1 = p \times n \times m$ multiplications,
- On a $\sum_{i=1}^{m}\sum_{j=1}^{n} 1 = n \times m$ additions.

Au total, on a $((p+1) \times n \times m)$ opérations, qui est moins que l'algorithme produit Matrice-Matrice "ijk" à 3 boucles. Donc, la complexité évolue en $o((p+1) \times n \times m)$ et $O(p \times n \times m)$.

Algorithme du produit Matrice-Matrice "ijk" à 1 boucle:

Le produit Matrice-Matrice "ijk" à 1 boucle implémenté dans Scilab est donné dans listing 1.4.

Listing 1.4: Produit Matrice-Matrice "ijk" avec 1 boucle

```
function[C]=matmat1b(A, B)

%Fonction produit Matrice-Matrice "ijk" a 1 boucle

function[C]=matmat1b(A, B)
[m p] = size(A) %Taille de la matrice A
[p n] = size(B) %Taille de la matrice B
C=zeros(m,n);   %Initiation de la matrice C


for i = 1:m
            C(i,:)=A(i,:)*B+C(i,:);
end

endfunction
funcprot(0)
```

Etude de Complexité:

On se base sur le nombre d'opérations dans le calcul de la complexité, comme suit:

- On a $\sum_{i=1}^{m} n \times p = m \times p \times n$ multiplications,
- On a $\sum_{i=1}^{m} n \times (p-1) = m \times (p-1) \times n$ additions.

Au total, on a $((2p-1) \times n \times m)$ opérations. Donc, la complexité évolue en $o((2p-1) \times n \times m)$.

3/ **Temps de calcul des algorithmes pour différentes tailles de matrices:**

Les variations du temps en fonction de n pour les 3 algorithmes du produit Matrice-Matrice "ijk" sont regroupés dans le tableau 1.2. Il est évident que l'algorithme du produit Matrice-Matrice "ijk"à 1 boucle prend beaucoup moins du temps (compléxité réduite).

IT.	m	p	n	Temps(s)-3boucles	Temps(s)-2boucles	Temps(s)-1boucle
1	3	2	3	0.00013	0.000156	0.000558
2	100	80	90	0.923494	0.036103	0.002565
3	300	200	280	23.934692	0.44034	0.047085
4	500	400	480	139.92913	1.902311	0.24971
5	800	700	780	619.32445	7.099506	1.144587
6	900	800	880	948.47501	10.344888	1.63069
7	1000	900	980	1294.0011	13.98408	2.360478

Table 1.2: Temps des algorithmes en fonction des tailles de matrices A et B.

Implementations de tests dans Scilab:

Les tests des produits Matrice-Matrice "ijk" sont donnés dans les listings 1.5, 1.6 et 1.7.

Listing 1.5: Test: Produit Matrice-Matrice "ijk" avec 1 boucle

```
%Test du Produit Matrice-Matrice "ijk" avec 1 boucle
A=rand(1000,900); %Matrice Aleatoire A (m*p)
B=rand(900,980); %Matrice Aleatoire B (p*n)

tic();
[C]=matmat1b(A, B)
toc();
disp(toc()); %printing the calculation time
```

Listing 1.6: Test: Produit Matrice-Matrice "ijk" avec 2 boucles

```
%Test du Produit Matrice-Matrice "ijk" avec 2 boucles
A=rand(1000,900); %Matrice Aleatoire A (m*p)
B=rand(900,980); %Matrice Aleatoire B (p*n)

tic();
[C]=matmat2b(A, B)
toc();
disp(toc()); %printing the calculation time
```

Listing 1.7: Test: Produit Matrice-Matrice "ijk" avec 3 boucles

```
%Test du Produit Matrice-Matrice "ijk" avec 3 boucles
A=rand(10,5); %Matrice Aleatoire A (m*p)
B=rand(5,6); %Matrice Aleatoire B (p*n)

tic();
[C]=matmat3b(A, B)
toc();
disp(toc()); %printing the calculation time
```

La figure 1.2 illustre la variation de temps de calcul en fonction des tailles de matrices A et B.

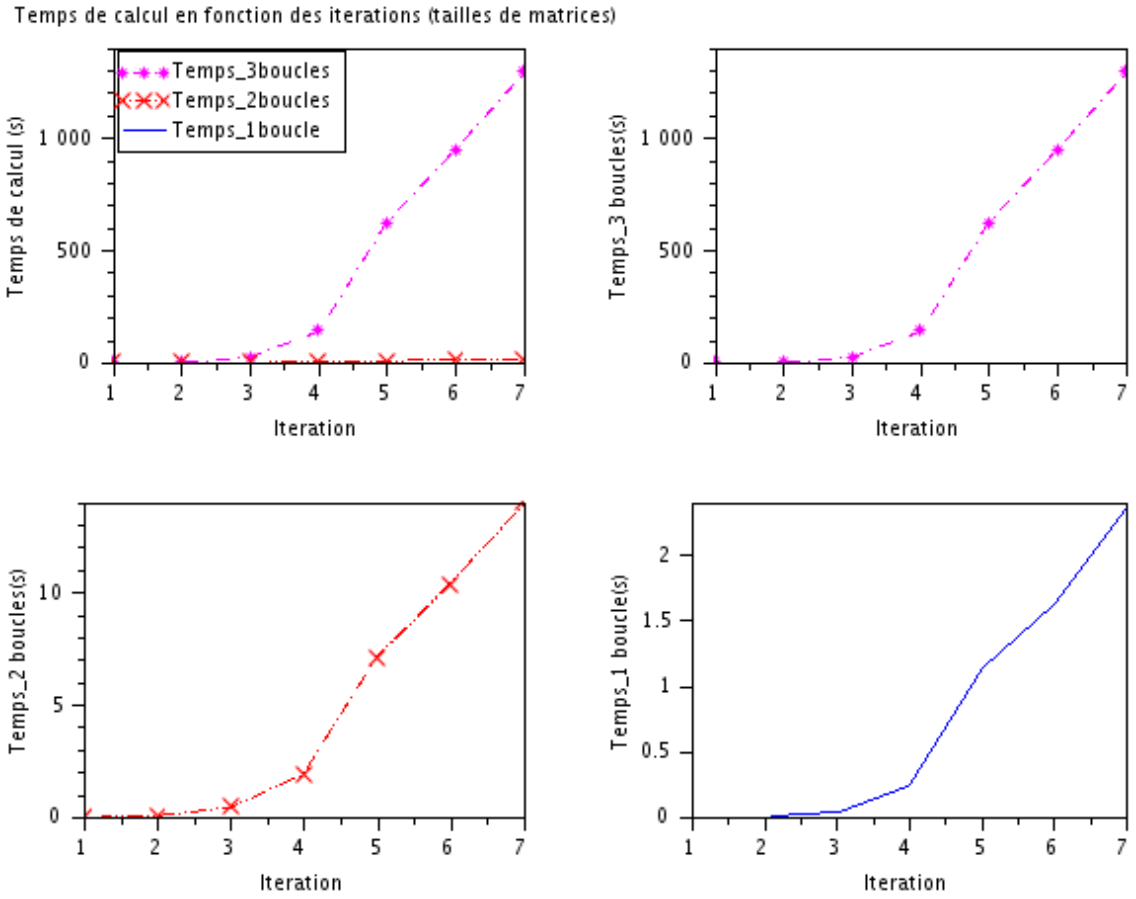

Figure 1.2: Temps de calcul des algorithmes Produit Matrice-Matrice pour différentes tailles de matrices A et B.

4/ D'après la figure 1.2, on remarque que le temps de calcul est plus important pour l'algorithme Produit Matrice-Matrice à 3 boucles. En effet, le temps de calcul de l'algorithme Produit Matrice-Matrice à 2 boucles est moins important que celui à 3 boucles et plus important que celui à 1 boucle. D'autre part, le temps de calcul augmente exponentiellement en fonction des tailles de matrices A et B. Cela est prévisible car le temps de calcul est proportionnel à la compléxité de l'algorithme.

Chapter 2

TP3 de Calcul Numérique

Les exercices 2, 3 et 4 du TP3 sont réalisés sur scilab. Ce compte rendu comprend l'analyse des algorithmes à implémenter, les observations et analyses des algorithmes, ainsi que des mesures de performances.

2.1 Exercice 2: Système triangulaire

1/ **Algorithmes de résolution par remontée et descente (Alg 10 et 11, Dufaud 2021):**
Soient U et L deux matrices triangulaires supérieure et inférieure de tailles $n \times n$.
La fonction usolve(U,b) dans listing 2.1 illustre l'algorithme de la méthode de remontée, qui consiste à résoudre le système linéaire triangulaire supérieure $U \times x = b$.

Listing 2.1: Méthode de remontée

```
%Alg10: Methode de remontee : resolution de Ux = b (version avec
%produit scalaire)

function[x]=usolve(U,b) %U: Matrice triangulaire superieure, b est un vecteur

 n = size(b)(1) %n= longueur du 1er element du vecteur b
 x=zeros(n, 1); %initialisation du vecteur des inconnus x

%**Methode de Remontee: On commence par l'inconnu x(n) et on monte a x(1)**
    x(n)=b(n)/U(n, n);  %La n-eme equation depend uniquement de l'inconnu x(n)
    for i = n-1:-1 : 1 % Resolution des (n-1) equations restantes
         x(i)=(b(i)-U(i,(i+1):n)*x((i+1):n))/U(i,i);
    end

endfunction
funcprot(0)
```

La fonction lsolve(L,b) dans listing 2.2 illustre l'algorithme de la méthode de descente, qui consiste à résoudre le système linéaire triangulaire inférieure $L \times x = b$.

Listing 2.2: Méthode de descente

```
%Methode de descente : resolution de Lx = b (version avec
%produit scalaire)

function[x]=lsolve(L,b) %L: Matrice triangulaire superieure, b est un vecteur

 n = size(b)(1) %n= longueur du 1er element du vecteur b
 x=zeros(n, 1); %initialisation du vecteur des inconnus x

%**Methode de Descente: On commence par x(1) et on descend a x(n)**
    x(1)=b(1)/L(1, 1); %La 1ere equation depend uniquement de l'inconnu x(1)
    for i = 2:n % Resolution des (n-1) equations restantes
      x(i)=(b(i)-L(i, 1:(i-1))*x(1:(i-1)))/L(i,i);
    end

endfunction
funcprot(0)
```

Etude de Complexité:

Complexité de l'algorithme de remontée: (Complexité quadratique (polynomiale))

On se base sur le nombre d'opérations dans le calcul de la complexité, comme suit:

- Pour n: On a 1 seule division
- Pour $i = (n-1) : -1 : 1$: On a:
  - $(n-i)$ multiplications dans la boucle, i.e. $\sum_{i=1}^{n-1}(n-i) = n \times (n-1) - \frac{n \times (n-1)}{2}$ multiplications,
  - $(n-i-1)$ additions dans la boucle, i.e. $\sum_{i=1}^{n-1}(n-i-1) = (n-1)^2 - \frac{n \times (n-1)}{2}$ additions,
  - 1 soustraction dans la boucle, i.e. $(n-1)$ soustractions,
  - 1 division dans la boucle, i.e. $(n-1)$ divisions.

Au total, on a n^2 opérations. Donc la complexité évolue en $O(n^2)$.

Complexité de l'algorithme de descente: (Complexité quadratique (polynomiale))

On se base sur le nombre d'opérations dans le calcul de la complexité, comme suit:

- Pour $n = 1$: On a 1 seule division
- Pour $i = 2 : n$: On a:

- $(i-1)$ multiplications dans la boucle, i.e. $\sum_{i=2}^{n}(i-1) = \frac{(n-1)\times(n+2)}{2} - (n-1)$ multiplications,
- $(i-2)$ additions dans la boucle, $\sum_{i=2}^{n}(i-2) = \frac{(n-1)\times(n+2)}{2} - 2\times(n-1)$ additions
- 1 soustraction dans la boucle, i.e. $(n-1)$ soustractions,
- 1 division dans la boucle, i.e. $(n-1)$ divisions.

Au total, on a n^2 opérations. Donc la complexité évolue en $O(n^2)$.

2/ La matrice A et le vecteur b générés dans le TP2, exercice 7 sont introduites dans les fonctions test de usolve et lsolve, présentées dans les listings 2.3 et 2.4.

Listing 2.3: Test du Méthode de remontée

```
%Test de l'exercice 7 du TP2 "Jouet"

A=[0.5819606, 0.6603468, 0.8009798; 0.9860136, 0.5958275, 0.5262907; 0.249952,
    0.5903676, 0.1496625]; %Matrice A (3*3) de l'ex Jouet, TP2
b= [0.847947; 0.6737939; 0.3919569]; %Vecteur b (3*1) de l'ex Jouet, TP2

U= triu(A); %Matrice triangulaire superieure U a partir du matrice A
[x]=usolve(U,b); %solution du systeme triangulaire superieur Ux=b
disp(x);
funcprot(0)
```

Listing 2.4: Test du Méthode de descente

```
%Validation de l'exercice 7 du TP2 "Jouet"

A=[0.5819606, 0.6603468, 0.8009798; 0.9860136, 0.5958275, 0.5262907; 0.249952,
    0.5903676, 0.1496625]; %Matrice A (3*3) de l'ex Jouet, TP2
b= [0.847947; 0.6737939; 0.3919569]; %Vecteur b (3*1) de l'ex Jouet, TP2

L=tril(A); %Matrice triangulaire inferieure L a partir du matrice A
[x]=lsolve(L,b); %solution du systeme triangulaire inferieur Lx=b
disp(x);
funcprot(0)
```

Les fonctions triu(A) et tril(A) permettent d'extraire des matrices triangulaires supérieure et inférieure de A, respectivement. On vérifie que la matrice U obtenue est une matrice triangulaire supérieure, comme suit:

$U =$

0.5819606	0.6603468	0.8009798
0.	0.5958275	0.5262907
0.	0.	0.1496625

La résolution du système $U \times x = b$ donne le vecteur $x \in R^{3\times1}$ suivant:

-0.8058118

-1.1824381

2.6189386

D'autre part, on vérifie que la matrice L obtenue est une matrice triangulaire inférieure:

$L =$

0.5819606 0. 0.

0.9860136 0.5958275 0.

0.249952 0.5903676 0.1496625

La résolution du système $L \times x = b$ donne le vecteur $x \in R^{3\times1}$ suivant:

1.4570522

-1.2803696

5.2361314

On remarque que la matrice L est la matrice A déduite des coefficients au dessus de la diagonale. Tandis que la matrice U est la matrice A déduite des coefficients au dessous de la diagonale.

On peut également tester le temps de calcul des deux fonctions usolve et lsolve.

Le tableau 2.1 indique la moyenne de ce temps sur 10 essais. Comme la complexité des algorithmes de la descente et de la remontée est en $O(n^2)$, donc le temps d'exécution des deux algorithmes est comparable.

Temps de Calcul	
usolve(U,b)	**lsolve(L,b)**
0.0002578s	0.0002151s

Table 2.1: Temps de calcul des fonctions usolve et lsolve.

2.2 Exercice 3: Gauss

1/ **Algorithme de résolution par élimination de Gauss sans pivotage (Alg 12, Dufaud 2021):**

L'algorithme de Gauss presenté dans listing 2.5 consiste à trouver la matrice M inversible, tel que $M \times A$ soit triangulaire.

$$A \times x = b \iff M \times A \times x = M \times b$$

Comme le produit $M \times A$ est triangulaire (élimination de Gauss), on peut alors utiliser l'algorithme de remontée. La triangularisation s'effectue en $(n-1)$ étapes et consiste à annuler les éléments des colonnes situés sous la diagonale principale, Dufaud 2021.

Listing 2.5: Résolution par élimination de Gauss sans pivotage

```
%Resolution par elimination de Gauss sans pivotage (Alg 12 p27)

function[x]=gausskij3b(A,b)

n = size(b)(1) %n

for k = 1:n-1 %Triangularisation de A en (n-1) etapes
    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) %Resolution Ax=b par remontee: Appelation de l'Alg10 (usolve)
endfunction
funcprot(0)
```

Etude de Complexité: (Complexité cubique (polynomiale))

On se base sur le nombre d'opérations dans le calcul de la complexité, comme suit:

- On a comme nombre d'opérations:
  - On a $\sum_{k=1}^{n-1}(\sum_{i=k+1}^{n} 1)$ divisions,
  - On a $\sum_{k=1}^{n-1}(\sum_{i=k+1}^{n}(1+\sum_{j=k+1}^{n} 1))$ soustractions,
  - On a $\sum_{k=1}^{n-1}(\sum_{i=k+1}^{n}(1+\sum_{j=k+1}^{n} 1))$ multiplications.

Au total, on a $(\frac{2n^3}{3}+\frac{n^2}{2}-\frac{7n}{6})$ opérations. Donc la complexité de l'algorithme évolue en $O(\frac{2n^3}{3})$. Comme la résolution du système linéaire $A \times x = b$ par la méthode de l'élimination de Gauss sans pivotage fait une appélation à la méthode de résolution $A \times x = b$ par remontée, alors la compléxité totale sera dans ce cas: $(\frac{2n^3}{3}+\frac{n^2}{2}-\frac{7n}{6})+n^2 = \frac{2n^3}{3}+\frac{3n^2}{2}-\frac{7n}{6}$

2/ **Test et Validation:** L'algorithme de résolution par élimination de Gauss sans pivotage a été testé sur des petites matrices aléatoires de tailles $n \times n$ (Listing 2.6).

Listing 2.6: Test de la Résolution par élimination de Gauss sans pivotage

```
%Test de l'algorithme de resolution par elimination de Gauss sans pivotage

A=rand(3,3); %Matrice A (3*3)
xg=rand(3,1); %vecteur xg (3*1)
b=A*xg; %vecteur b (3*1)

%Temps de calcul
tic();
[x]=gausskij3b(A,b) %Resolution par elimination de Gauss sans pivotage
disp(toc());

%Erreurs avant et arriere
err=norm((xg-x)/xg,2); %Calcul de l'erreur avant
relres=norm(b-A*x,2)/(norm(A,2)*norm(x,2)); %Calcul de l'erreur arriere
disp(err); %Erreur avant
disp(relres); %Erreur arriere

funcprot(0)
```

Les moyennes sur 10 mesures du temps de calcul, ainsi que les erreurs avant et arrière en fonction de n ont été regroupés dans le tableau 2.2. La matrice A est générée à chaque fois en utilisant la fonction $rand(n, n)$ de Scilab. Puis, un vecteur colonne $xg \in R^{n\times 1}$ est généré avec la fonction $rand(n, 1)$. Le vecteur b étant le produit de A et xg. Cela va être utile dans le calcul de l'erreur avant et arrière.

$$\begin{cases} erreur\ avant = ||\frac{xg-x}{xg}|| \\ erreur\ arrière = \frac{||b-A\times x||}{||A||\times||x||} \end{cases}$$

n	Temps de Calcul-avg	Erreur Avant-avg	Erreur Arrière-avg
3	0.0002065	1.060506e-15	1.41104e-16
5	0.0002738	3.008471e-14	1.51621e-16
7	0.0003798	1.83807e-14	3.9467e-16
10	0.0005263	1.30081e-14	5.0508e-16
15	0.0013905	8.99728e-14	1.37593e-15
20	0.0029381	4.93003e-13	1.309092e-14
50	0.0397174	8.72489e-12	5.4439e-15
100	0.7440034	2.95123e-11	3.67006e-14

Table 2.2: Variation du temps de calcul, erreurs avant et arrière en fonction de n.

Pour bien illustrer la variation de ces quantités en fonction de n, les courbes montrants la variation du temps de calcul, ainsi que les erreurs avant et arrière en fonction de n sont dessinées à l'aide de Scilab, comme montré dans les figures 2.1 et 2.2.

On en déduit que les temps de calcul, les erreurs avant et arrière augmentent en fonction de n.

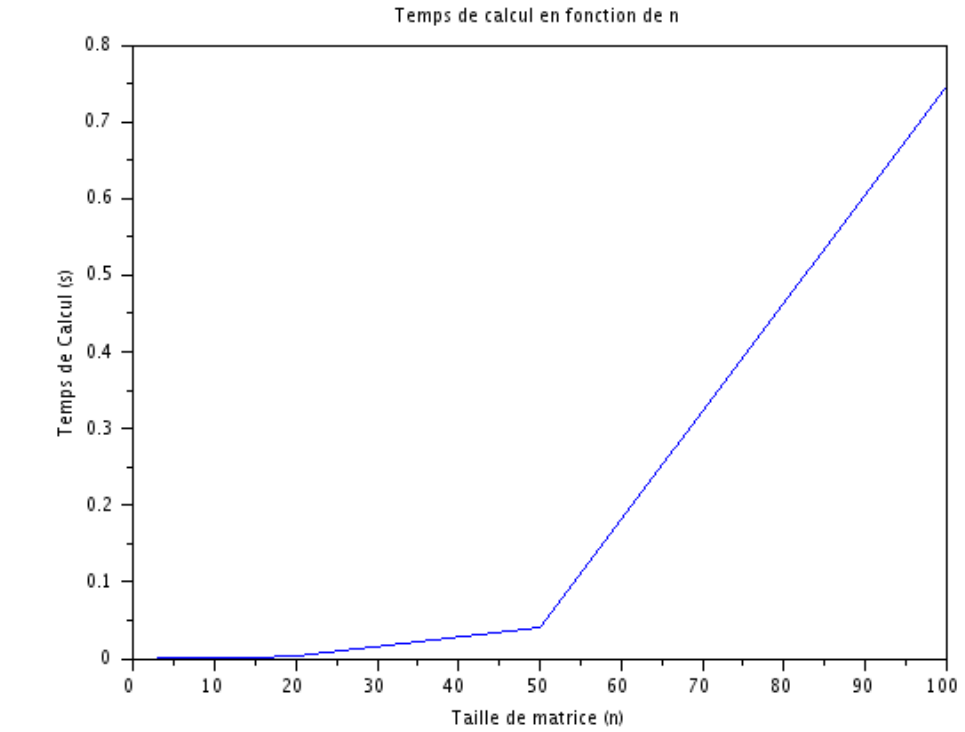

Figure 2.1: Variation de la moyenne du temps de calcul en fonction de la taille de matrice n.

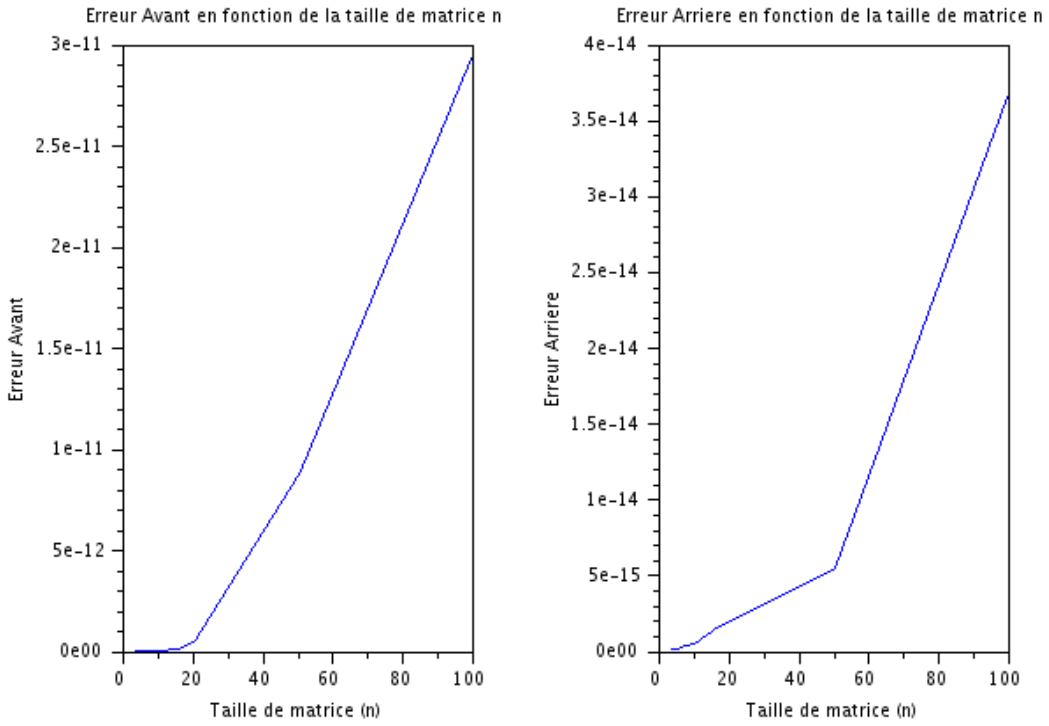

Figure 2.2: Variation de la moyenne des erreurs avant et arrière en fonction de n.

2.3 Exercice 4: *LU*

1/ Algorithme de factorisation *LU* (Alg 13, Dufaud 2021):

Pour résoudre un système linéaire $A \times x = b$, on fait recourt aux étapes suivantes (listing 2.7):

- Triangularisation de la matrice A à l'aide de l'algorithme d'élimination de Gauss,
- Résolution du système triangulaire $L \times y = b$ par la méthode de la descente,
- Résolution du système triangulaire $U \times x = y$ par la méthode de la remontée.

La matrice A est le produit du matrice triangulaire supérieure U et inférieure L:

$$A = L \times U$$

Listing 2.7: Élimination de Gauss et écriture compacte de LU (version scalaire a 3 boucles "kij")

```
%Elimination de Gauss et ecriture compacte de LU (version
%scalaire    3 boucles "kij") (Alg 13 p30)

function [L,U]=mylu3b(A)
n=size(A,1); %Taille de la matrice

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

L = tril(A, -1)+eye(n,n); %matrice L triangulaire inferieure avec des 1 sur la
    diagonale

U = triu(A); %matrice u triangulaire superieur

endfunction
funcprot(0)
```

Etude de Complexité: (Complexité cubique (polynomiale))
On se base sur le nombre d'opérations dans le calcul de la complexité, comme suit:

- On a comme nombre d'opérations:
  - On a $\sum_{k=1}^{n-1} \sum_{i=k+1}^{n} 1 = \frac{n(n-1)}{2}$ divisions,
  - On a $\sum_{k=1}^{n-1} \sum_{i=k+1}^{n} \sum_{j=k+1}^{n} 1 = \frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6}$ soustractions,
  - On a $\sum_{k=1}^{n-1} \sum_{i=k+1}^{n} \sum_{j=k+1}^{n} 1 = \frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6}$ multiplications.

Au total, on a $(\frac{2n^3}{3} - \frac{n^2}{2} - \frac{n}{6})$ opérations. Donc la complexité de l'algorithme évolue en $O(\frac{2n^3}{3})$.

2/ L'algorithme précédent est testé pour des petites matrices A, comme montré dans le tableau 2.3. L'erreur commise de la factorisation $L \times U$ est definie comme: $err = ||A - L \times U||$.

Les valeurs sont obtenues en moyennant à chaque fois 10 mesures.

Taille n du matrice A	**Temps de calcul-avg**	**Erreur de Factorisation-avg**
3	9.14e-05	1.04412e-16
5	0.0001877	2.9044e-16
7	0.0003686	3.8499e-16
10	0.0008811	1.1106e-15
15	0.0021304	2.60088e-15
20	0.0044774	1.43627e-14
50	0.0570981	1.8172e-14
100	0.4545931	1.16529e-13

Table 2.3: Erreurs commises de factorisation en fonction des tailles n de matrices A.

La variation du temps de calcul et de l'erreur de factorisation en fonction de n est illustrée dans la figure 2.3. On remarque que le temps de calcul et de l'erreur de factorisation augmentent en fonction de la taille de matrice n.

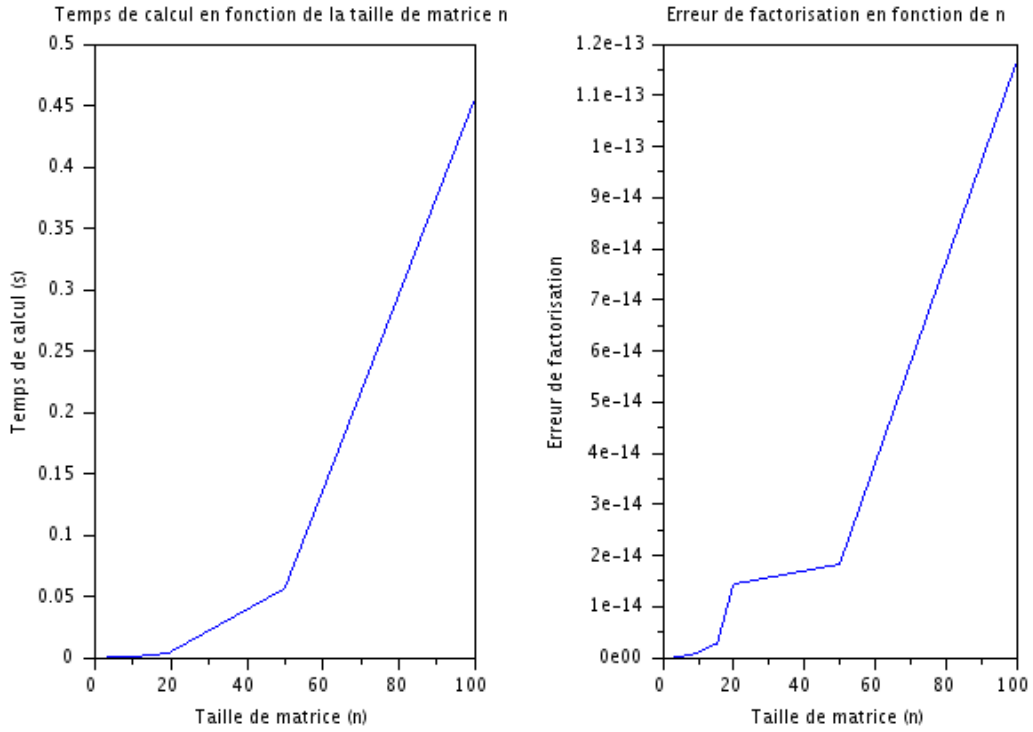

Figure 2.3: Variation de la moyenne de temps de calcul et de l'erreur de factorisation en fonction de taille de matrice n.

3/**Algorithme de factorisation $L \times U$ à une boucle:**

L'algorithme de factorisation $L \times U$ à une boucle est presenté dans listing 2.8.

Listing 2.8: Algorithme de factorisation $L \times U$ à une boucle

```
%LU Optimisee a une boucle

function [L,U]=mylu1b(A)
[n n]=size(A);

for k = 1:n-1

    A( k+1:n,k)=A( k+1:n,k)/A(k,k);
    A(k+1:n, k+1:n)=A(k+1:n,k+1:n)-A(k+1:n,k)*A(k,k+1:n);

end
L = tril(A,-1)+eye(n,n); %matrice L triang. inf. avec des 1 sur la diagolanle

U = triu(A); %matrice u triangulaire superieur

endfunction
funcprot(0)
```

Etude de Complexité: (Complexité cubique (polynomiale))

- On a comme nombre d'opérations:
  - On a $\sum_{k=1}^{n-1}(n-k) = \frac{n(n-1)}{2}$ divisions,
  - On a $\sum_{k=1}^{n-1}(n-k) = \frac{n(n-1)}{2}$ soustractions,
  - On a $\sum_{k=1}^{n-1}(n-k)^2 = \frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6}$ multiplications.

Au total, on a $(\frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6})$ opérations. Donc la complexité de l'algorithme évolue en $O(\frac{n^3}{3})$, c'est à dire la moitié de l'algorithme de factorisation $L \times U$ classique dans listing 2.7.

Ainsi, l'algorithme de factorisation $L \times U$ est optimisé.

4/ **Méthode de pivot partiel:**

La méthode de pivot partiel à une complexité en $O(n^2)$. Elle consiste à prendre pour pivot le plus grand élément de la colonne en valeur absolue, Dufaud 2021. L'algorithme 2.9 garantit l'existence de la factorisation $L \times U$ d'une matrice A.

Listing 2.9: Factorisation LU avec la méthode de Pivot Partiel

```
function [L,U,P]=mylu(A)

n = size (A,1); %size n

P=eye(n,n); %initialization du matrice de permutation P
L=eye(n,n); %initialization du matrice L
U=A; %initialization du matrice U

for k=1:n %Algorithme du Pivot Partiel P*A = L*U
    [piv, ind] = max(abs(U(k:n,k))); %recuperation du max piv a la ligne ind du
     vecteur colonne sous l element diagonal A(k,k)

    ind = k-1+ind; %conversion de l indice local en un indice global de la
    ligne a echanger

    if (ind ~= k) then %interchangement des lignes ind et k

    %on echange les lignes ind et k dans la matrice U
        new = U(k, :); %stockage temporaire de la ligne a echanger dans le
    vecteur new
        U(k, :) = U(ind, :);  %echange
        U(ind, :) = new;

 %on echange les lignes ind et k dans la matrice de permutation P
        new1 = P(k, :); %stockage temporaire de la ligne a echanger dans le
    vecteur new1
        P(k, :) = P(ind, :);  %echange
        P(ind, :) = new1;

    %On echange les lignes lignes ind et k dans les colonnes 1:k-1 de la
    matrice L
        if k>=2
            new2 = L(k,1:k-1);
            L(k,1:k-1) = L(ind,1:k-1);
            L(ind,1:k-1) = new2;
        end
    end

    for i=k+1:n
     L(i,k)= U(i,k)/U(k,k);
     U(i,:)=U(i,:)-L(i,k)*U(k,:);
    end
end

endfunction
funcprot(0)
```

5/ La méthode de pivot partiel est testée et comparée pour différentes tailles de matrice n à la fonction lu() de Scilab, comme montré dans listing 2.10. Les résultats sont données dans le tableau 2.4.

On note par:

- **err1:** l'erreur entre la matrice L obtenue par l'algorithme du pivot partiel et celle obtenue par la fonction lu() de Scilab,

- **err2:** l'erreur entre la matrice U obtenue par l'algorithme du pivot partiel et celle obtenue par la fonction lu() de Scilab,
- **err3:** l'erreur entre la matrice de permutation P obtenue par l'algorithme du pivot partiel et celle obtenue par la fonction lu() de Scilab.

On moyenne les résultats sur 5 mesures des erreurs err1, err2 et err3. Ainsi que le temps de calcul pour l' algorithme de pivot partiel et la fonction lu() de Scilab.

Listing 2.10: Comparison de l'algorithme de Pivot Partiel avec la fonction lu() de Scilab

```
A=rand(3,3); %Matrice A

tic();
[L,U,P]=mylu(A);
toc();
disp(toc()); %display le temps de calcul

disp(L); %Display la matrice L
disp(U); %Display la matrice U
disp(P); %Display la matrice P

%Verification des resultats
T1= L*U;
T2=P*A;

disp(T1); %Display le produit matriciel L*U
disp(T2); %Display le produit matriciel P*A

%Validation en utilisant la fonction lu() de Scilab
tic();
[L1,U1,P1]= lu(A); %fonction lu() de Scilab
toc();
disp(toc());

disp(L1); %Display la matrice L1 de la fonction lu() de Scilab
disp(U1); %Display la matrice U1 de la fonction lu() de Scilab
disp(P1); %Display la matrice P1 de la fonction lu() de Scilab

%Calcul des erreurs
err1 = norm(L-L1); %calcul d'erreur entre L de l'algo et L1 de Scilab
err2 = norm(U-U1); %calcul d'erreur entre U de l'algo et U1 de Scilab
err3 = norm(P-P1); %calcul d'erreur entre P de l'algo et P1 de Scilab

funcprot(0)
```

On trouve qu' en testant l'algorithme de factorisation LU avec la methode de Pivot Partiel:

$$T1 = T2 \Longrightarrow P \times A = L \times U$$

La comparison entre l'algorithme de pivot partiel et la fonction lu() de Scilab en terme des matrices L, U et P générées, ainsi que le temps de calcul sont données dans la figure 2.4.

n	**err1**	**err2**	**err3**	**Temps-Algo**	**Temps-lu()Scilab**
3	2.0398e-16	6.585e-17	0	0.0001774	3.12e-05
5	2.54678e-16	1.020255e-16	0	0.0003156	3.5e-05
10	1.10898e-15	8.2172e-16	0	0.0007488	2.96e-05
15	1.46674e-15	1.5306e-15	0	0.0014262	3.3e-05
20	1.9102e-15	1.9868e-15	0	0.0022444	3.88e-05
50	1.0471e-14	2.2024e-14	0	0.009152	9.6e-05

Table 2.4: Comparison de l'algorithme de Pivot Partiel avec la fonction lu() de Scilab.

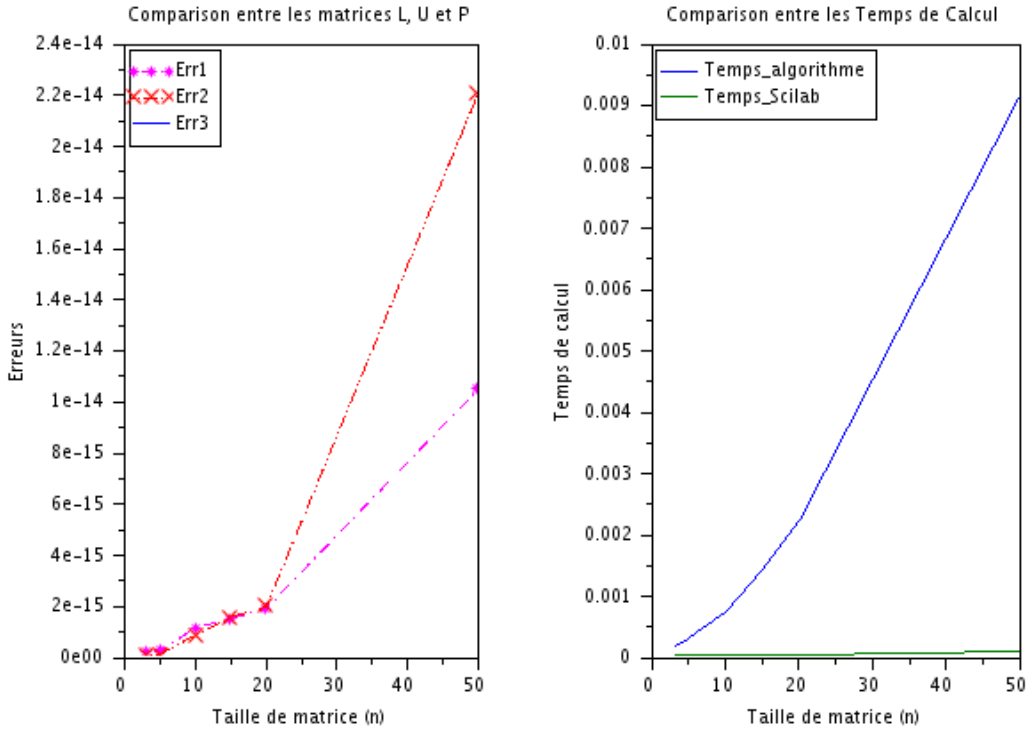

Figure 2.4: Comparison de l'algorithme de Pivot Partiel avec la fonction lu() de Scilab.

La différence entre les matrices L, U et P de l'algorithme de Pivot Partiel implementé et celles de la fonction lu() de Scilab a un ordre de 10^{-15} et augmente légérement en fonction de la taille de la matrice A. D'autre part, le temps de l'exécution de la fonction lu() de Scilab est faible devant celui de l'algorithme de Pivot Partiel.

6/**Optionnel: Matrice de T.Davis**

Pour lire les matrices creuses de T.Davis, il faut d'abord installer le module Matrix Market, *"Overview of sparse matrices in Scilab"* 2020. On considère l'exemple d'une matrice creuse de taille 130×130.

Dans listing 2.11, la fonction mylu est appliquée au matrice de T. Davis. Pour utiliser l'algorithme

classique du Pivot Partiel sans faire une transformation pour la format de stockage particulière de la matrice creuse, il est nécessaire de transformer cette matrice creuse en une matrice complète en utilisant la fonction full() de Scilab. Mais, cela ne permet pas de profiter de la particularité de la matrice creuse pour optimiser le temps de calcul, Fu et al. 1998.

Listing 2.11: Algorithme de Pivot Partiel sur une matrice creuse de T.Davis

```
%Sparse Matrix
umfdir = fullfile(SCI,"modules","umfpack","examples");
filename = fullfile(umfdir,"arc130.rua");
[C] = ReadHBSparse(SCI+"/modules/umfpack/demos/arc130.rua");

%Converting the sparse matrix to a full one
A = full(C)

%Methode de Pivot Partiel
[L,U,P]=mylu(A)

funcprot(0)
```

References

"Overview of sparse matrices in Scilab" (2020). Scilab. URL: https://wiki.scilab.org/Overview%20of%20sparse%20matrices%20in%20Scilab.

"What Is Backward Error?" (2020). Applied Mathematics, numerical linear algebra and software. URL: https://nhigham.com/2020/03/25/what-is-backward-error/.

Dufaud, Thomas (2021). *"Cours CHPS M1: Calcul Numérique - Algèbre linéaire dense"*.

Fu, Cong, Jiao, Xiangmin, and Yang, Tao (1998). "Efficient sparse LU factorization with partial pivoting on distributed memory architectures". In: *IEEE Transactions on Parallel and Distributed Systems* 9.2, pp. 109–125.

Oliveira, Pablo (2021). *"Cours CHPS M1: Arithmétique à virgule flottante"*.

Appendix A

Appendix Chapter

Les TPs 2 et 3 du calcul numérique sont déposés dans le dépot git **"TPs-TDs-Calcul-Numerique-CHPS-M1-"**. Le code SSH de ce dépot est le suivant:

git@github.com:Chaichas/TPs-TDs-Calcul-Numerique-CHPS-M1-.git