Rapport Master M1

Aicha Maaoui

Master Calcul Haute Performance Simulation (CHPS)

TP4-TP5 Calcul Numerique

Décembre 2021

Institut des Sciences et Techniques des Yvelines (ISTY)

Abstract

Le TP 4 "Exploitation des Structures et Calcul Creux" est associé au cours de Calcul Numérique "Vers l'optimisation d'algorithmes numériques et algèbre creuse".
Il a pour but:

- Implémentation de l'algorithme de la factorisation LDL^T pour une matrice symétrique,
- Implémentation de l'algorithme de la factorisation Cholesky LL^T pour une matrice symétrique définie positive,
- Comparison des algorithmes de factorisation LDL^T et LU,
- Implémentation des algorithmes de stockage COO, CSR et CSC pour les matrices creuses,
- Algorithme de calcul du transposée d'une matrice creuse,
- Algorithme de calcul du Produit Matrice-Vecteur creux pour les formats COO et CSR,
- Test et validation, étude de compléxité, ainsi que des mesures de performance des algorithmes implémentés.

Le TP 5 "Application à l'équation de la chaleur 1D stationnaire" est associé au cours de Calcul Numérique "Vers l'optimisation d'algorithmes numériques et algèbre creuse". Il a pour objectif la résolution d'un système linéaire obtenu par la discrétisation (méthode des différences finies) de l'équation de la chaleur 1D stationnaire.

Il comporte:

- Travail préliminaire et explication du cas test Poisson 1D,
- Explication de l' architecture du code,
- Explication des appels aux bibliothèques externes,

- Explication du format de stockage et illustration des stockages en Row Major et Col Major,
- Explication et validation des appels à dgbsv,
- Explication et validation des appels à dgbmv,
- Implémentation de l'algorithme de factorisation LU pour la matrice tridiagonal (au format GB en C),
- Implémentation Scilab des algorithmes Jaccobi, Gauss Seidel et Richardson.

Contents

List of Tables

Listings

Chapter 1

TP4 de Calcul Numérique

Les exercices 1.b, 4 et 5 sont réalisés sur scilab. Ce compte rendu comprend l'analyse des algorithmes à implémenter et les mesures de performances.

1.1 Exercice 1.b: Factorisation LDL^T de A symétrique

Cet exercice tient en compte des matrices particulières: (i) symétrique dans le cas de la factorisation LDL^T, (ii) symétrique définie positive dans le cas de la factorisation LL^T.
Dans le cas d'une matrice particulière, on n'a pas besoin de stocker toute la matrice A. Ce qui permet de réduire l'occupation mémoire.
1/ **Algorithme de la factorisation LDL^T pour une matrice symétrique:**

Soit A une matrice inversible symétrique $\in R^{n\times n}$.
Il existe une unique factorisation LDL^T de la matrice A, tel que:

- D est une matrice diagonale; ie $D = diag(d_{11}, ..., d_{nn})$,
- L est une matrice triangulaire inférieure, avec des coefficients unitaires sur la diagonale (Unit Lower Triangular Matrix).

Dans un premier temps, on propose dans listing (1.1) l'algorithme de calcul de la matrice L et du vecteur d constitué par les coefficients diagonales de la matrice D.

Ainsi, la résolution du système $Ax = b$ s'effectue en trois étapes:

- résoudre le système $Ly = b$, où L est une matrice triangulaire inférieure tel que les coefficients sur la diagonale de L, $L(i, i) = 1$,

- résoudre le système $Dz = y$, où D est une matrice diagonale,
- résoudre le système $L^T x = z$, où L^T est une matrice triangulaire supérieure, transposée de la matrice L. Comme le système est triangulaire, on obtient x par la méthode de remontée.

Description de l'algorithme LDL^T:

On présente deux algorithmes pour la factorisation LDL^T de la matrice A, répartis comme suit:

- Algorithme standard pour la factorisation LDL^T de la matrice A (listing (1.1)),
- Algorithme avec forme compacte pour la factorisation LDL^T de la matrice A (listing (1.2)).

On suppose qu'on connait les premières colonnes $(j-1)$ de la factorisation $A = LDL^T$, comme illustré dans la figure (1.1).

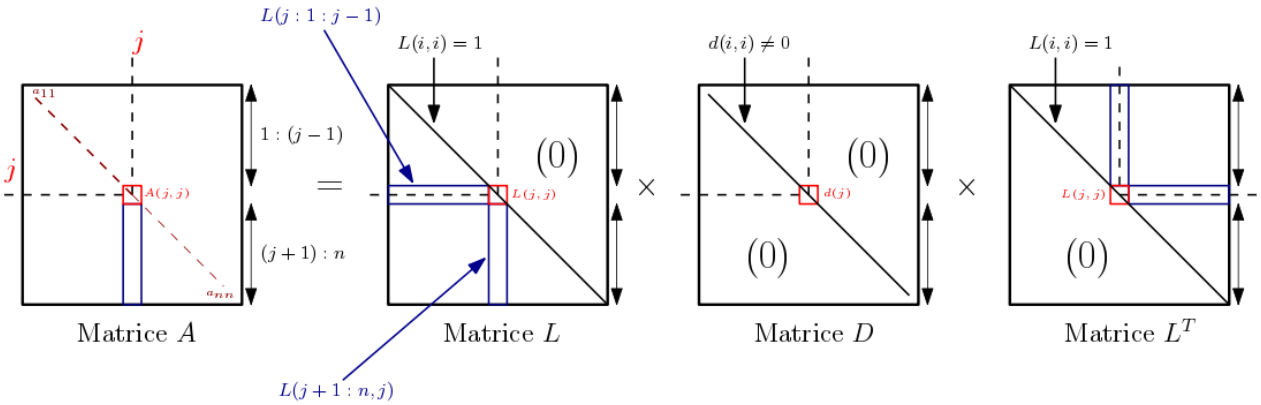

Figure 1.1: Description de l'algorithme de factorisation $A = LDL^T$.

Soit le vecteur colonne $v(1:j)$ solution du système triangulaire $A = L \times v$, avec:

$$v(1:j) = L(j, 1:j) \times d(1:j), \quad v(j) = d(j) \tag{1.1}$$

Ainsi, le vecteur colonne j de la matrice A pour les lignes de j à n s'écrit:

$$A(j:n, j) = L(j:n, 1:j) \times v(1:j) \tag{1.2}$$

Le vecteur $d(j)$ est obtenu de la formule du produit scalaire:

$$A(j,j) = d(j) + \sum_{k=1}^{j-1} d(k) \times L(j,k)^2 \tag{1.3}$$

On considère maintenant les $(n-j)$ equations restantes, qui permettent d'identifier $L(j+1:n, j)$:

$$A(j+1:n, j) = L(j+1:n, j) \times d(j) + L(j+1:n, 1:j-1) \times v(1:j-1)) \tag{1.4}$$

L'algorithme décrit est presenté dans listing (1.1). Il permet le calcul de la matrice triangulaire inférieure L, avec des coefficients unitaires sur sa diagonale, ainsi que les coefficients d_{ii} de la matrice diagonale D.

Listing 1.1: Factorisation LDL^T d'une matrice symétrique $A \in R^{n \times n}$

```
function [L,d]=LDLFactorisation(A)  %Algo. standard de la factorisation LDL'

n = size(A,1); %taille de la matrice A carre (n*n)

for i = 1:n
    L(i,i)=1; %Tous les coeffs de L sur la diagonale sont egaux a 1
end

d(1) = A(1,1); %1er element du vecteur D
L(2:n,1) = A(2:n,1)/d(1); %Coeffs de la 1ere colonne de la matrice L

for j=2:n
    for i=1:j-1
        v(i)= L(j,i)*d(i); %Calcul du vecteur v
    end

    %Calcul des elements du vecteur D
    d(j)= A(j,j)-L(j,1:j-1)*v(1:j-1);

    %Calcul des elements de L
    L(j+1:n,j)=(A(j+1:n,j)-L(j+1:n, 1:j-1)*v(1:j-1) )/d(j);
end

endfunction
funcprot(0)
```

Forme compact de la factorisation LDL^T de A:

La forme compacte de la factorisation $A = LDL^T$ est presentée dans listing (1.2). Elle consiste à écraser $A(i,j)$ par:

$$A(i,j) = \begin{cases} L(i,j), & \text{si } i > j \\ d(i), & \text{si } i = j \end{cases}$$

Cela nous permet de réduire davantage l'occupation mémoire lors de la factorisation LDL^T de la matrice A, comme on écrase les cases mémoire de A et on les remplace par L et d_i.
De plus, comme A est une matrice symétrique, l'occupation mémoire est évaluée à $\frac{n^2}{2}$.

Listing 1.2: Factorisation LDL^T d'une matrice symétrique $A \in R^{n\times n}$: Forme compacte

```
function [L,d]=LDLFactorisation_Compacte(A) %Forme compacte

n = size(A,1); %taille de la matrice A: Matrice carre (n*n)

A(1,1) = A(1,1);
A(2:n,1) = A(2:n,1)/A(1,1);

for j=2:n
    for i=1:j-1
        v(i)= A(j,i)*A(i,i); %Calcul du vecteur v
    end
    %Calcul des elements du vecteur D
    A(j,j)= A(j,j)-A(j,1:j-1)*v(1:j-1);
    %Calcul des elements de L
    A(j+1:n,j)=(A(j+1:n,j)-A(j+1:n, 1:j-1)*v(1:j-1) )/A(j,j);
end
%Obtention de d et L a partir de la matrice A
d = diag(A); %Coeff d de la matrice diagonale D
L = tril(A- diag(d),-1)+eye(n,n);

endfunction
funcprot(0)
```

Etude de complexité Théroqie de l'algorithme LDL^T:

- Le nombre d'opérations est évalué à:
  - $\sum_{j=1}^{n}(j-1)+(j-1)+(n-j)\times(j-1)$ multiplications,
  - $\sum_{j=1}^{n}1+(n-j)$ soustractions,
  - $\sum_{j=1}^{n}(j-2)+(n-j)\times(j-2)$ additions.
  - $\sum_{j=1}^{n}(n-j)$ divisions.

Au total, on a $(\frac{n^3}{3}+n^2-\frac{4n}{3})$ opérations. Donc la complexité de l'algorithme évolue en $O(\frac{n^3}{3})$ pour n qui tend vers l'infini, c'est à dire la moitié de l'algorithme de factorisation LU.

Test, validation et performance de l'algorithme LDL^T:

L'algorithme de factorisation $A = LDL^T$ sous sa forme standard (listing (1.1)) et sa forme compacte (listing (1.2)) sont testés comme indiqué dans listing (1.3).

Pour utiliser la fonction rand() de Scilab, il faut tout d'abord vérifier que $A \in R^{n\times n}$ soit symétrique ($A = A^T$). Une solution consiste à calculer la matrice $w = rand(n,n)$, puis on peut poser $A = w \times w^T$.

Listing 1.3: Test de Factorisation LDL^T d'une matrice symétrique $A \in R^{n \times n}$

```
W = rand(3,3); %Matrice qlq construite a partir de la fonction random
A = W*W'; %Matrice A symetrique construite a partir de w et son transposee

%Factorisation de LDL' (Algo Factorisation_LDL')
tic(); %Activation du temps
[L,d] = LDLFactorisation(A); %Factorisation de LDL'
toc(); %Temps de calcul

%Calcul de l erreur commise de la factorisation LDL' =0
err_LDL = norm(A-L*diag(d)*L');

%Factorisation compacte de LDL' (Algo Factorisation_Compacte_LDL')
tic(); %Activation du temps
[L_LDL_Compacte,d_LDL_Compacte]=LDLFactorisation_Compacte(A); %Factorisation
    compacte de LDL'
toc(); %Temps de calcul

%Calcul de l erreur commise de la factorisation compacte LDL' =0
err_LDL_Compacte = norm(A-L_LDL_Compacte*diag(d_LDL_Compacte)*L_LDL_Compacte');
```

Dans un premier temps, On va vérifier la validité des algorithmes de factorisation LDL^T.

Ainsi, la moyenne de l'erreur relative de factorisation $(A - L \times D \times L^T)$ est calculée, comme montré dans le tableau (1.1).

Taille n	**Erreur de la forme standard/ compacte**	**cond(A)**
3	5.551e-17	556.02688
5	2.220e-16	1461.0267
10	1.227e-15	3760.2106
15	2.381e-15	27897.335
20	3.626e-15	36612.426
40	1.184e-14	719820.87
60	5.861e-14	1380587.3
80	8.540e-14	2997920.3
100	8.869e-14	6838246.8
1000	1.411e-11	3.821D+09

Table 1.1: Erreur de la factorisation $A = LDL^T$.

On vérifie que l'erreur relative de la factorisation $(A - L \times D \times L^T)$ est faible, mais augemente en fonction de la taille de matrice n (de même cond(A) augmente).

L'erreur relative de factorisation $\delta = A - L \times D \times L^T$ est due aux erreurs d'arrondi de l'ordinateur, relative à la précision système de représenter des nombres réels en virgule flottante, et se produisent surtout quand la taille de la matrice n est grande: perte de chiffres significatifs par arrondi, *"Arrondis des ordinateurs, erreurs de mesure"* 2021.

Le traçage de la courbe de l'erreur relative de la factorisation LDL^T de la matrice symétrique A en fonction de sa taille n est illustré dans la figure (1.2).

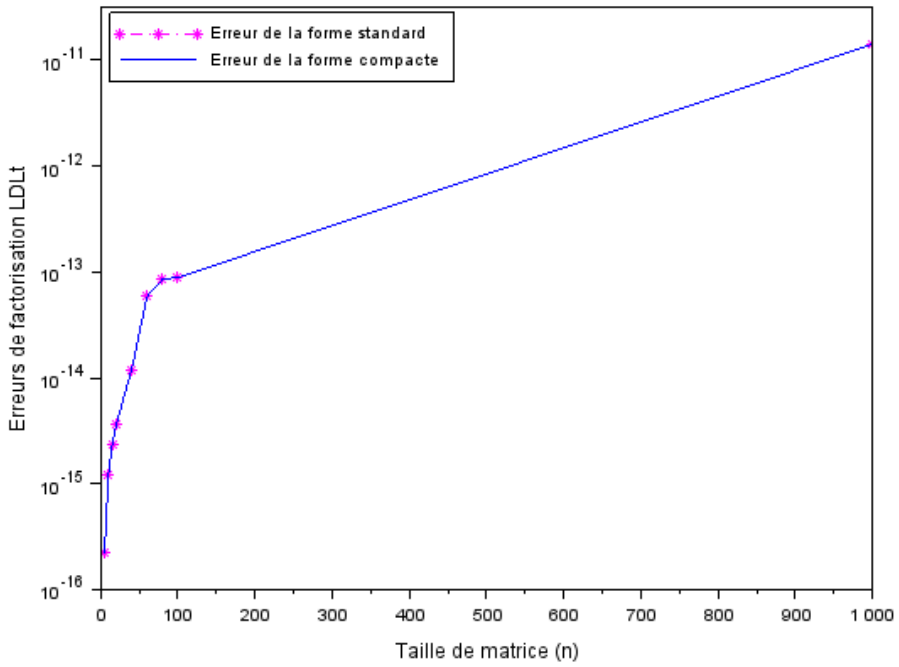

Figure 1.2: Erreurs relatives de factorisation $A = LDL^T$.

Dans la suite, on considère l'algorithme de factorisation LDL^T sous sa forme compacte. La moyenne du temps de calcul est ainsi évalué pour n allant de 3 à 1000, comme montré dans le tableau (1.2).

Taille n	**Temps de calcul (s)**
3	0.000081
5	0.0001050
10	0.00023
15	0.000307
20	0.000462
40	0.001527
60	0.0027040
80	0.0049450
100	0.0068600
1000	1.3096240

Table 1.2: Temps de calcul de la factorisation $A = LDL^T$ (Complexité Temporelle).

La figure (1.3) illustre la complexité temporelle de l'algorithme de factorisation $A = LDL^T$. On remarque que le temps de calcul évolue en fonction de la taille de matrice d'entrée $A \in R^{n \times n}$.

Il faut noter que le nombre d'opérations effectuées par seconde est indiqué par la fréquence du processeur. Dans le cas d'étude, la fréquence du processeur est évaluée à $2.8GHz = 2.8 \times 10^9 Hz$, i.e., 2.8 milliards d'opérations par seconde. Ainsi, une opération nécessite environ:

$T = \frac{1}{f} = \frac{1}{2.8 \times 10^9} = 0.357 \times 10^{-9} = 0.357$ nanoseconde. La complexité pour n grande est evaluée à $\frac{n^3}{3}$ opérations. Dans notre cas, pour $n = 1000$, le nombre d' opérations est evalué à: $\frac{1000^3}{3}$, soit environ $\frac{1000^3}{3} \times 0.357$ nanosecondes.

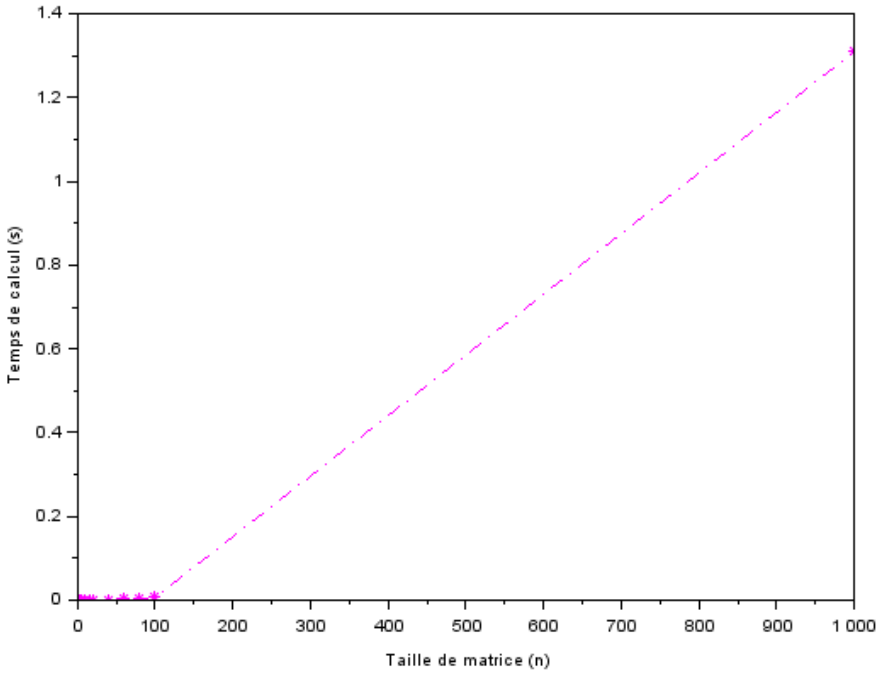

Figure 1.3: Complexité temporelle des algorithmes de factorisation $A = LDL^T$.

2/ **Algorithme de la factorisation de Cholesky LL^T:**

Soit A une matrice inversible symétrique définie positive $\in R^{n\times n}$.

Description de l'algorithme:

A admet une décomposition unique sous la forme $A = LL^T$, où L est une matrice triangulaire inférieure avec des éléments diagonaux positifs.

On suppose qu'on connait les premières colonnes $(j-1)$ de la factorisation $A = LL^T$, comme illustré dans la figure (1.4).

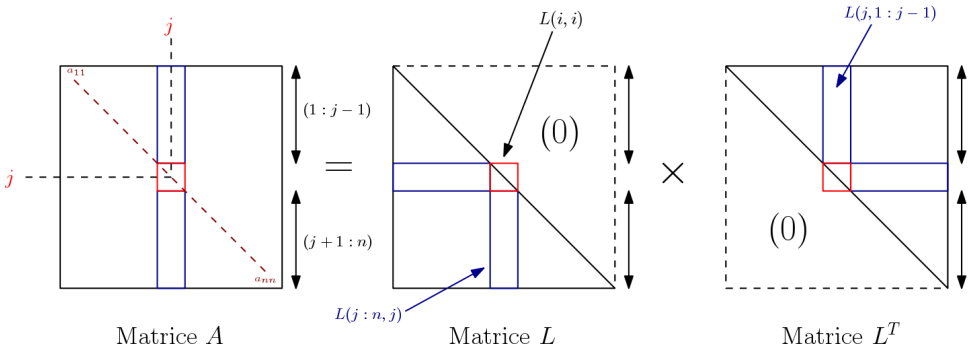

Figure 1.4: Description de l'algorithme de factorisation $A = LL^T$.

La factorisation LL^T de la matrice A s'écrit comme suit:

$$A = L \times L^T \Rightarrow \sum_{k=1}^{n} L(i,k) \times L(j,k) \tag{1.5}$$

Les élements $L(i,i)$ de la matrice triangulaire inférieure L sont donnés par:

$$L(i,i) = \sqrt{A(i,i) - \sum_{k=1}^{i-1} L(i,k) \times L(i,k)^T} \tag{1.6}$$

$$\text{Pour } \; i = 1 : \;\; L(1,1) = \sqrt{A(1,1)} \tag{1.7}$$

Pour la première colonne, les coefficients $L(2:n,1)$ de la matrice L s'écrivent:

$$L(2:n,1) = \frac{A(2:n,1)}{L(1,1)} \tag{1.8}$$

On détermine maintenat la ième colonne de L pour $2 \leq i \leq n$:

$$L(j,i) = \frac{(A(j,i) - \sum_{k=1}^{i-1} L(j,k) \times L(i,k)^T)}{L(i,i)} \tag{1.9}$$

L'algorithme de Cholesky LL^T est donné dans listing (1.4).

Listing 1.4: Factorisation Cholesky LL^T d'une matrice symétrique définie positive $A \in R^{n \times n}$

```
function [L] = cholesky(A)

n=size(A,1); %Taille de la matrice A: n*n

L(1,1) = sqrt(A(1,1)); %Calcul du 1er element de la matrice L
L(2:n,1) = A(2:n,1)/L(1,1); %Calcul des elements de la 1ere colonne de L

for i = 2:n %boucle sur les lignes

    %Calcul des coefficients sur la diagonale de L
    L(i,i) = sqrt(A(i,i)-L(i,1:i-1)*L(i,1:i-1)');

    for j = i+1:n %boucle sur les colonnes
        L(j,i) = (A(j,i)-L(j,1:i-1)*L(i,1:i-1)')/L(i,i); %Calcul des autres
    coeffs de la matrice L
    end
end
endfunction
funcprot(0)
```

Test et validation de l'algorithme LL^T:

Soit la matrice $A \in R^{4\times4}$ suivante, symétrique ($A^T = A$) définie positive (déterminants des sous matrices de A sont strictement positives):

$$\text{A}= \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 5 & 5 & 5 \\ 1 & 5 & 14 & 14 \\ 1 & 5 & 14 & 15 \end{pmatrix}$$

On obtient la matrice L triangulaire inférieure obtenue suivante:

$$\text{L}= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 \\ 1 & 2 & 3 & 0 \\ 1 & 2 & 3 & 1 \end{pmatrix}$$

On peut vérifier ce resultat avec la fonction chol() de Scilab, qui retourne une matrice triangulaire supérieure. Ainsi, on vérifie bien que: $L = chol(A)'$ et $A = L \times L^T$

La factorisation Cholesky d'une matrice symétrique définie positive A est fréquemment utilisée pour la résolution des équations normales dans le cas où $ker(A) = 0$, i.e. A^TA est symétrique définie positive, *"Algèbre Linéaire Numérique"* 2006.
Elle permet aussi la réduction de l'occupation mémoire vu qu'on calcule uniquement la matrice L.

Etude de complexité Théorique de l'algorithme LL^T:

- Le nombre d'opérations est évalué à:
  - n racines carrées,
  - $\sum_{i=1}^{n}(n-i)$ divisions,
  - $\sum_{i=1}^{n}(n+1)(n-i) - \sum_{i=1}^{n}\sum_{j=i+1}^{n} j$ multiplications,
  - $\sum_{i=1}^{n}(n+1)(n-i) - \sum_{i=1}^{n}\sum_{j=i+1}^{n} j$ soustractions.

Au total, on a $(\frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6})$ opérations. Donc la complexité de l'algorithme évolue en $\text{O}(\frac{n^3}{3})$ pour n qui tend vers l'infini, c'est à dire la moitié de l'algorithme de factorisation LU.

3/ **Comparison entre la factorisation LDL^T et LU de A:**

Comparison de la complexité théorique:

Factorisation LU	**Factorisation LDL^T**	**Factorisation LL^T**
$\frac{2n^3}{3}$	$\frac{n^3}{3}$	$\frac{n^3}{3}$

Table 1.3: Comparison de la complexité asymptotique entre les factorisation LU, LDL^T et LL^T.

D'après le tableau (1.3), la complexité des algorithmes de factorisations LDL^T et Cholesky LL^T est la moitié de celle de LU. Ainsi, LDL^T et LL^T sont plus stable que l'algorithme de factorisation LU.

Comparison de la complexité temporelle: Temps de calcul

Le tableau (1.5) montre la variation du temps de calcul obtenue par Scilab des algorithmes de factorisations LDL^T et LU en fonction de la taille de matrice A.

Taille n	LU	LDL^T**, forme compacte**
5	0.000348	0.000211
10	0.001003	0.00032
20	0.004469	0.000632
40	0.036776	0.002045
60	0.1153720	0.003558
80	0.279546	0.005799
100	0.50551	0.0083650
1000	515.38789	1.194137

Table 1.4: Comparison entre temps de calcul de (Complexité Temporelle).

La variation du temps de calcul en fonction de la taille n de la matrice A est illustrée dans la figure (1.5).

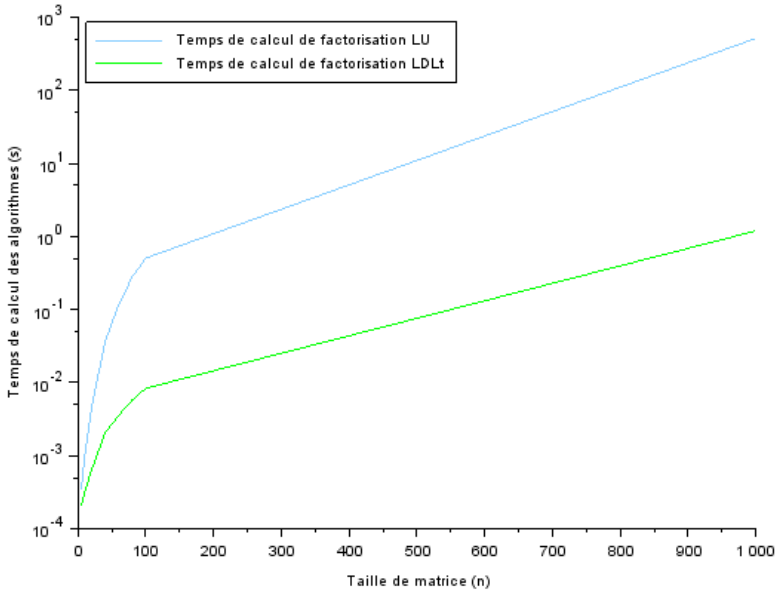

Figure 1.5: Comparison de la complexité temporelle des algorithmes de factorisation d'une matrice A symétrique.

Le tableau (1.5) montre la variation des erreurs de factorisation obtenue par Scilab des algorithmes de factorisations LDL^T et LU en fonction de la taille n de la matrice symétrique A.

Taille n	LU	LDL^T, **forme standard**	**cond(A)**
5	2.220D-16	1.110D-16	693.28446
10	1.386D-15	1.047D-15	1790.7644
20	6.647D-15	4.119D-15	67704.804
40	2.383D-14	1.185D-14	218188.22
50	3.261D-14	1.733D-14	186190.89
80	8.602D-14	7.544D-14	18920139
100	1.253D-13	8.147D-14	6.194e+10

Table 1.5: Comparison entre erreur de factorisation.

On peut aussi dessiner la courbe montrant la variation des erreurs relatives de factorisation en fonction de la taille n, comme presenté dans la figure (1.6). On conclut ainsi que l'erreur relative est plus faible dans le cas de la factorisation de LDL^T que la factorisation LU. Ainsi, le résultat est plus stable face aux erreurs d'arrondi.

La courbe du conditionnement de la matrice A en fonction de sa taille n montre que ce dernier augmente en fonction des imprécisions.

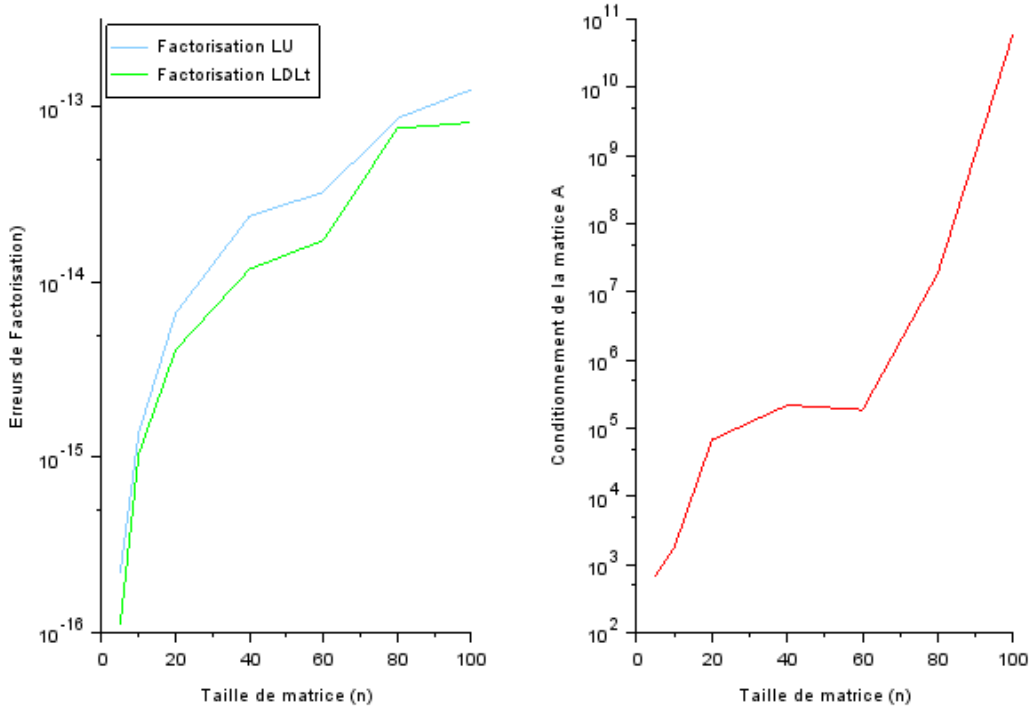

Figure 1.6: Comparison de l'erreur de factorisation des algorithmes LU et LUL^T.

4/ **Discussion:**

- Le calcul du conditionnement de la matrice A est important dans la résolution numérique des systèmes linéaires $Ax = b$, car il joue le rôle d'un indicateur de stabilité numérique pour les méthodes directes (plus le conditionnement est grand, plus la solution numérique est différente de celle exacte) et de vitesse de convergence pour les méthodes itératives, *"Algèbre Linéaire Numérique"* 2006. Ainsi, le conditionnement de la matrice borne l'erreur relative à partir de la solution obtenue,

- L'intéret d'utiliser des matrices diagonales est le fait qu'elles sont toujours inversibles, tandis que l'utilisation des matrices triangulaires facilite la résolution numérique à l'aide des algorithmes de remontée et descente,

- Les algorithmes de factorisation LDL^T et de Cholesky sont plus stables que celui de factorisation LU, car la complexité évolue en $O(\frac{n^3}{3})$. Donc, deux fois moins que la complexité de factorisation LU qui évolue en $O(\frac{2n^3}{3})$,

- Au terme de l'occupation mémoire, on peut bénéficier de la nature symétrique de la matrice A, utilisée dans la factorisation LDL^T. Ainsi, on stockera $\frac{n^2}{2}$ au lieu de n^2,

- La forme $A = LDL^T$ permet la détermination de l'inertie de la matrice A (*"Loi d'inertie de Sylvester"* 2021),

- La nature du problème envisagé impose la nature de la matrice A. Par exemple, dans le cas des problèmes de thermique ou d'élasticité, on a recours à des matrices symétriques définies positives, *"Chapitre 3 Systèmes d'équations"* 2015. Il s'agit donc de la factorisation LL^T de Cholesky. La factorisation LDL^T est restreinte au cas où la matrice A est positive, tandis que la factorisation LU est utilisée pour des matrices carrées A de formes générales (avec la matrice A et ses sous-matrices principales inversibles).

1.2 Exercice 4: Stockage CSR et CSC

0/ **Stockage des matrices creuses:** Pour des matrices avec des éléments égaux à zéro, on peut faire recourt au différentes formats de stockage caractéristiques des matrices creuses.
Soit nnz le nombre des éléments non nuls d'une matrice creuse $A \in R^{m \times n}$. On peut citer comme formats de stockage des éléments non nuls d'une matrice creuse A:

- **COO (Coordinate Storage):** On stocke les éléments non nuls d'une matrice creuse dans $AA \in R^{1 \times nnz}$ (selon les lignes). Pour l'accès aux éléments, les indices sur les lignes i et sur les colonnes j sont stockées dans $JA \in R^{1 \times nnz}$ et $IA \in R^{1 \times nnz}$, respectivement,
- **CSR (Compressed Sparse Matrix):** On stocke les éléments non nuls d'une matrice creuse dans $AA \in R^{1 \times nnz}$. Pour l'accès aux éléments, on stocke dans $JA \in R^{1 \times nnz}$ le numéro j de la colonne respective a l'élément et dans $IA \in R^{1 \times (n+1)}$ les pointeurs sur les lignes.
- **CSC (Compressed Sparse Column):** On stocke les éléments non nuls d'une matrice creuse dans $AA \in R^{1 \times nnz}$ (selon les colonnes). Pour l'accès aux éléments, on stocke dans $JA \in R^{1 \times nnz}$ le numéro i de la ligne respective a l'élément et dans $IA \in R^{1 \times (m+1)}$ les pointeurs sur les colonnes.

1/ Soit la matrice $A \in R^{8 \times 6}$ comme suit:

$$\text{A}= \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 & 0 & 0 \\ 0 & 11 & 3 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 & 25 & 7 \\ 0 & 0 & 28 & 0 & 0 & 0 & 0 & -2 \end{pmatrix}$$

On a $nnz = 12$ éléments non nuls de la matrice A.

Stockage CSR de A:

AA	15	22	-15	11	3	2	-6	91	25	7	28	-2
JA	1	4	6	2	3	7	4	1	7	8	3	8
IA	1	4	7	8	8	11	13	-	-	-	-	-

Table 1.6: Stockage CSR de la matrice A, exo.4.

2/ Stockage CSC de A:

AA	15	19	11	3	28	22	-6	-15	2	25	7	-2
JA	1	4	2	2	5	1	3	1	2	4	4	5
IA	1	3	4	6	8	8	9	11	13	-	-	-

Table 1.7: Stockage CSC de la matrice A, exo.4.

3/ **Transposée de la matrice creuse A:**

Desciption de l'algorithme: Soit la matrice $A \in R^{6\times 8}$. la transposée de la matrice A est $A^T \in R^{8\times 6}$. On considère le stockage COO. Soit la matrice triplet $\in R^{12\times 3}$, dont la repartition des colonnes est la suivante:

- **Première colonne:** On stocke les indices des lignes des éléments non nuls de A, i,
- **Deuxième colonne:** On stocke les indices des colonnes des éléments non nuls de A, j,
- **Troisième colonne:** On stocke les éléments non nuls de A.

Soit la matrice triplet obtenue à partir de la matrice A:

$$\text{triplet}= \begin{pmatrix} 1 & 1 & 15 \\ 1 & 4 & 22 \\ 1 & 6 & -15 \\ 2 & 2 & 11 \\ 2 & 3 & 3 \\ 2 & 7 & 2 \\ 3 & 4 & -6 \\ 5 & 1 & 91 \\ 5 & 7 & 25 \\ 5 & 8 & 7 \\ 6 & 3 & 28 \\ 8 & 6 & -2 \end{pmatrix}$$

Listing 1.5: Algorithme pour calculer le triplet d'une matrice creuse A

```
%Extract Triplet from matrix A
function [triplet]=Triplet(A)

[m n]=size(A); %taille de la matrice A (m*n)
nnz = 0; %nnz= elements non nuls de la matrice A

    for i=1:m
        for j=1:n
            if (A(i,j)~=0) then %condition: element non nul de la matrice A
                nnz = nnz+1; %nombre des elements non nuls augmente de 1
                %1ere colonne de triplet (indices i des lignes)
                triplet(nnz,1) = i;
                 %2eme colonne de triplet (indices j des colonnes)
                triplet(nnz,2) = j;
                %3eme colonne de triplet (elements non nuls de la matrice A)
                triplet(nnz,3) = A(i,j);
            end
        end
    end
endfunction
funcprot(0)
```

L'étape suivante consiste à inverser la première et la deuxième colonne dans triplet. Soit la matrice triplet' $\in R^{12\times 3}$, tel que:

$$\text{triplet'}= \begin{pmatrix} 1 & 1 & 15 \\ 4 & 1 & 22 \\ 6 & 1 & -15 \\ 2 & 2 & 11 \\ 3 & 2 & 3 \\ 7 & 2 & 2 \\ 4 & 3 & -6 \\ 1 & 5 & 91 \\ 7 & 5 & 25 \\ 8 & 5 & 7 \\ 3 & 6 & 28 \\ 8 & 6 & -2 \end{pmatrix}$$

Pour obtenir la matrice transposée de A à partir de triplet', il faut organiser cette dernière selon les valeurs minimales des lignes de la première colonne (tri). Si deux valeurs de lignes sont égales, on compare le minimum de la deuxième colonne. Soit la matrice tripletInvSor le résultat de cet algorithme. Alors:

$$\text{tripletInvSor}= \begin{pmatrix} 1 & 1 & 15 \\ 1 & 5 & 91 \\ 2 & 2 & 11 \\ 3 & 2 & 3 \\ 3 & 6 & 28 \\ 4 & 1 & 22 \\ 4 & 3 & -6 \\ 6 & 1 & -15 \\ 7 & 2 & 2 \\ 7 & 5 & 25 \\ 8 & 5 & 7 \\ 8 & 6 & -2 \end{pmatrix}$$

L'algorithme qui permet de calculer la tranposée d'une matrice creuse A est donné dans listing (1.6).

Listing 1.6: Algorithme pour calculer la tranposée d'une matrice creuse A

```
%Extract Triplet from matrix A to proceed
function [tripletInvSort]=SortedInverseTriplet(triplet)

nnz = size(triplet,1); %nnz= nombre des elements non nuls de la matrice triplet
temp = zeros(nnz,3); %Initialisation de la matrice temp
%Calcul du transposee de la matrice Triplet
    for i=1:nnz %de 1 a nnz= nbre des elements non nuls
        for j=1:3
            temp = triplet(i,1); %stockage de la 1ere colonne de triplet
            triplet(i,1)=triplet(i,2); %inverser les colonnes 1 et 2
            triplet(i,2)=temp; %triplet(i,2)=triplet(i,1)
        end
    end
%Sorting du Transposee de la matrice Triplet
max= triplet(1,1); %inialisation de la valeur max de triplet
    for i=1:(nnz-1)
            if (triplet(i,1) < triplet(i+1,1)) then
                max = triplet(i+1,1); %Recherche de la valeur max de triplet
            end
    end
k=1; %initialisation du scalaire k
    for i=1:max
        for j=1:nnz
            if(triplet(j,1)==i) then
                tripletInvSort(k,1) = triplet(j,1); %sorting ligne1
                tripletInvSort(k,2) = triplet(j,2); %sorting ligne2
                tripletInvSort(k,3) = triplet(j,3); %sorting ligne3
                k=k+1; %increment k de 1
            end
        end
    end
endfunction
funcprot(0)
```

Le programme écrit en Scilab est testé avec la matrice A de l'exercice 4, comme indiqué dans listing (1.7).

Listing 1.7: Tranposée de la matrice creuse A de l'exerice 4 et validation avec Scilab

```
%Matrice A de lexercice 4
A=[15,0,0,22,0,-15,0,0;0,11,3,0,0,0,2,0;0,0,0,-6,0,0,0,0;0,0,0,0,0,0,0,0;
91,0,0,0,0,0,25,7;0,0,28,0,0,0,0,-2];

[triplet]=Triplet(A); %Triplet de la matrice A, listing (1.5)

[tripletInvSort]=SortedInverseTriplet(triplet); %Inverse Triplet organise de la
    matrice A, listing 1.2
disp(tripletInvSort); %display tripletInvSort

%Validation avec Scilab --> Validated
A1=sparse (A); %obtention de la matrice creuse A1 a partir de A
At=A1'; %Tranposee de la matrice creuse A
disp(At); %Display Tranposee de la matrice creuse A
```

Avec la fonction sparse(A) de Scilab, on obtient la matrice transposée de A suivante (sous la forme "(i,j) val"):

(1, 1) 15.
(1, 5) 91.
(2, 2) 11.
(3, 2) 3.
(3, 6) 28.
(4, 1) 22.
(4, 3) -6.
(6, 1) -15.
(7, 2) 2.
(7, 5) 25.
(8, 5) 7.
(8, 6) -2.

Il est à noter que la transposée d'une matrice creuse A avec la format de stockage CSR est une matrice avec la format de stockage CSC.

4/ La complexité de l'algorithme de calcul de la tranposée d'une matrice A est: $O(3 \times nnz)$.

Le tableau 1.3 représente la moyenne de 10 mesures du temps de calcul de l'algorithme tranposée de la matrice creuse A de l'exercice.

Matrice	**Moyenne du temps de calcul**
$A \in R^{8\times6}$	0.0002152 s

Table 1.8: Moyenne de temps de calcul de l'algorithme tranposée de A.

1.3 Exercice 5: Produit Matrice-Vecteur Creux

Préambule: Pour optimiser le stockage mémoire d'une matrice creuse A, ainsi que le temps de calcul (performance), les éléments non nuls de la matrice sont uniquement stockés. Ainsi, la complexité devient O(nnz) au lieu de O($m \times n$), avec nnz est le nombre des éléments non nuls de la matrice creuse et $m \times n$ est la taille de la matrice. On propose dans (0) les différents formats de stockage d'une matrice creuse: COO, CSR et CSC.

Le but de cet exercice est d' ulitiser les formats COO et puis CSR dans le calcul du produit matrice creuse $A \in R^{m \times n}$ et vecteur $x \in R^{n \times 1}$, comme suit:

$$A \times x = y \in R^{m \times 1}$$

0/ **Les algorithmes de Stockages:** Les algorithmes des différents formats de stockages d'une matrice creuse A sont donnés dans les listings de (1.9) à (1.11). Ils se basent sur le calcul de nnz, le nombre des éléments non nuls d'une matrice creuse A (listing (1.8)).

Listing 1.8: Eléments non nuls d'une matrice creuse A

```
function [nnz]=NNZ_SPMat(A) %Calcul des elements ~=0 d'une matrice creuse A
[m n]=size(A); %taille de la matrice creuse A (m*n)
nnz = 0; %initialisation, nnz= elements non nuls de la matrice creuse A

    for i=1:m
        for j=1:n
            if (A(i,j)~=0) then %element non nul
                nnz = nnz+1; %incrementation du nombre des elements ~=0 de A
            end
        end
    end
endfunction
funcprot(0)
```

Listing 1.9: Stockage COO d'une matrice creuse A

```
function [AA,JA,IA]=COO_SPMat(A) %Stockage COO d'une matrice creuse A
[m n]=size(A);  %taille de la matrice A (m*n)
[nnz]=NNZ_SPMat(A); %nombre des elements non nuls d'une matrice creuse A
AA=zeros(nnz,1); %initialisation du vecteur AA (nnz*1)
JA=zeros(nnz,1); %initialisation du vecteur JA (nnz*1)
IA=zeros(nnz,1); %initialisation du vecteur IA (nnz*1)
k=1; %initialisation du compteur k
    for i=1:m
        for j=1:n
            if (A(i,j)~=0) then %element non nul de A
                AA(k) = A(i,j); %remplissage du vecteur AA: elements ~=0 de A
                JA(k) = i; %remplissage du vecteur JA: i des elements ~=0 de A
                IA(k) = j; %remplissage du vecteur IA: j des elements ~=0 de A
                k=k+1; %incrementation de k
            end
        end
    end
endfunction
funcprot(0)
```

Listing 1.10: Stockage CSR d'une matrice creuse A

```
%Stockage CSR d'une matrice creuse A
function [AA,JA,IA]=CSR_SPMat(A)

[m n]=size(A);  %taille de la matrice A (m*n)
[nnz]=NNZ_SPMat(A); %nombre des elements non nuls d'une matrice creuse A
AA=zeros(nnz,1); %initialisation du vecteur AA (nnz*1)
JA=zeros(nnz,1); %initialisation du vecteur JA (nnz*1)
IA=zeros(n+1,1); %initialisation du vecteur IA (n+1*1)
k=1; %initialisation du compteur k
    for i=1:m
        for j=1:n
            if (A(i,j)~=0) then %element non nul de A
                AA(k) = A(i,j); %remplissage du vecteur AA (selon les lignes)
                JA(k) = j; %remplissage du vecteur JA: j des elements ~=0 de A
                k=k+1; %incrementation de k
            end
        end
    end
%Remplissage du vecteur IA
nnz1=1; %initialisation du compteur sur les nbres des elements non nuls
IA(1)=1; %1er element du vecteur IA
IA(n+1)=nnz+1; %Element (n+1) du vecteur IA
    for i=1:m
        for j=1:n
            if (A(i,j)~=0) then %element non nul de A
                nnz1=nnz1+1; %nombre des elements non nuls de A
            end
        end
IA(i+1)=nnz1; %Remplissage du vecteur IA par des pointeurs sur les lignes
    end
endfunction
funcprot(0)
```

Listing 1.11: Stockage CSC d'une matrice creuse A

```
%Stockage CSC d'une matrice creuse A
function [AA,JA,IA]=CSC_SPMat(A)

[m n]=size(A);  %taille de la matrice A (m*n)
[nnz]=NNZ_SPMat(A); %nombre des elements non nuls d'une matrice creuse A
AA=zeros(nnz,1); %initialisation du vecteur AA (nnz*1)
JA=zeros(nnz,1); %initialisation du vecteur JA (nnz*1)
IA=zeros(m+1,1); %initialisation du vecteur IA (m+1*1)
nnz1=1; %initialisation du compteur sur les lignes
k=1; %intialisation du compteur k
IA(1)=1; %1er element du vecteur IA
IA(n+1)=nnz+1; %Element (n+1) du vecteur IA
    for j=1:n %boucle sur les lignes
        for i=1:m %boucle sur les colonnes
            if (A(i,j)~=0) then %elements non nuls de A
                AA(k) = A(i,j); %vecteur AA (selon les colonnes)
                JA(k) = i; %vecteur JA: i des elements ~=0 de A
                k=k+1; %incrementation de k
                nnz1=nnz1+1; %incrementation de nnz1
            end
        end
IA(j+1)= nnz1; %Remplissage du vecteur IA par des pointeurs sur les colonnes
    end
endfunction
funcprot(0)
```

On peut également tester les différentes formats de stockage d'une matrice creuse A et la comparer avec les fonctions de Scilab pour une matrice creuse, comme montré dans listing (1.12).

Listing 1.12: Tests et Validation des algorithmes de stockage d'une matrice creuse A

```
%Test des formats de stockage d'une matrice creuse A: COO, CSR et CSC

A=[1,2,0,11,0;0,3,4,0,0;0,5,6,7,0;0,0,0,8,0;0,0,0,9,10]; %matrice creuse A(5*5)

%Stockage COO
[AA_COO,JA_COO,IA_COO]=COO_SPMat(A); %Algo COO

%Stockage COO dans Scilab
%Pour tester le stockage COO on utilise space(A) --> format(JA(i),IA(i),AA(i))
D=sparse(A); %(JA(i),IA(i),AA(i)) --> Algo validated

%On peut aussi utiliser [ij,AA_COO1,mn]=spget(D), ou D = matrice creuse de A
[ij,AA_COO1,mn]=spget(D); % COO avec Scilab

%Stockage CSR
[AA_CSR,JA_CSR,IA_CSR]=CSR_SPMat(A); %Algo CSR

%Stockage CSC
[AA_CSC,JA_CSC,IA_CSC]=CSC_SPMat(A); %Algo CSC

%Test avec Scilab de CSC
B=sparse(A);
[IA_CSC1,JA_CSC1,AA_CSC1]=sp2adj(B); %CSC avec Scilab

%Erreurs entre l'algo CSC et Scilab:
Err_AA = norm(AA_CSC-AA_CSC1); %Erreur entre AA de l'algo implemente et Scilab
Err_JA = norm(JA_CSC-JA_CSC1); %Erreur entre JA de l'algo implemente et Scilab
Err_IA = norm(IA_CSC-IA_CSC1); %Erreur entre IA de l'algo implemente et Scilab
disp(Err_AA); %Erreur =0 --> Algo validated
disp(Err_JA); %Erreur =0 --> Algo validated
disp(Err_IA); %Erreur =0 --> Algo validated

A1=[1,2,0,11,0;0,3,4,0,0;0,5,6,7,0]; %matrice A(5*3)
[AA_COO,JA_COO,IA_COO]=COO_SPMat(A1); %COO
[AA_CSR,JA_CSR,IA_CSR]=CSR_SPMat(A1); %CSR
[AA_CSC,JA_CSC,IA_CSC]=CSC_SPMat(A1); %CSC
```

Dans la suite, on va illustrer les trois formats de stockage COO, CSR et CSC de la matrice A considerée dans le test de listing (1.12).

$$\text{A}= \begin{pmatrix} 1 & 2 & 0 & 11 & 0 \\ 0 & 3 & 4 & 0 & 0 \\ 0 & 5 & 6 & 7 & 0 \\ 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 9 & 10 \end{pmatrix}$$

Le nombre des éléments non nuls de la matrice A est $nnz = 11$.

Considérons dans un premier lieu le stockage COO de la matrice creuse A.

On regroupe les éléments non nuls de la matrice A selon les lignes dans le tableau AA, qui sera de taille $1 \times nnz$. Chaque élément a un indice ligne i (donné dans le tableau JA) et un indice colonne j (donné dans le tableau IA), comme montré dans la figure (1.7).

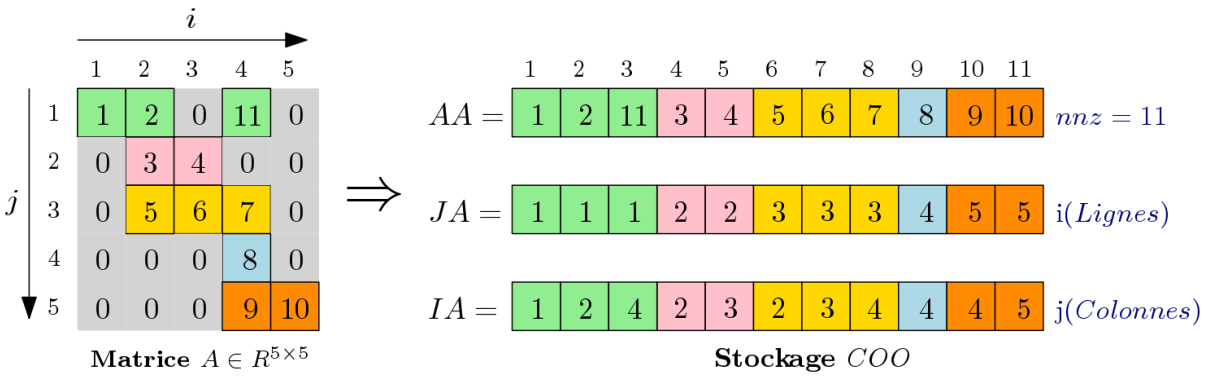

Figure 1.7: Stockage COO de la matrice A.

Considérons dans un deuxième lieu le stockage CSR de la matrice creuse A.

On regroupe les éléments non nuls de la matrice A selon les lignes dans le tableau AA, qui sera de taille $1 \times nnz$. On stocke dans le tableau JA les indices j des numéros de colonnes associées, et dans le tableau IA, qui est de taille $1 \times (n+1)$, les pointeurs sur les lignes, comme montré dans la figure (1.8).

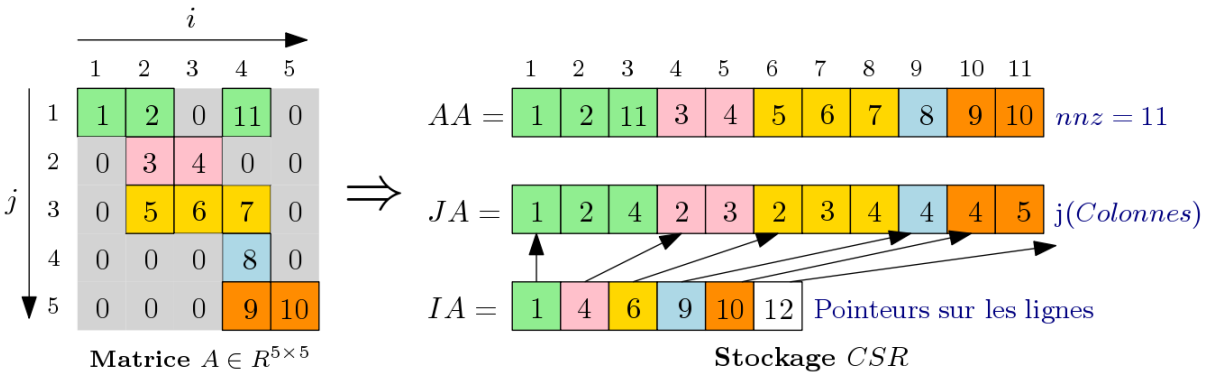

Figure 1.8: Stockage CSR de la matrice A.

Finalement, on considère le stockage CSC de la matrice creuse A.

On regroupe les éléments non nuls de la matrice A selon les colonnes dans le tableau AA, qui sera de taille $1 \times nnz$. On stocke dans le tableau JA les indices i des numéros de lignes associées, et dans le tableau IA, qui est de taille $1 \times (m+1)$, les pointeurs sur les colonnes, comme montré dans la figure (1.9).

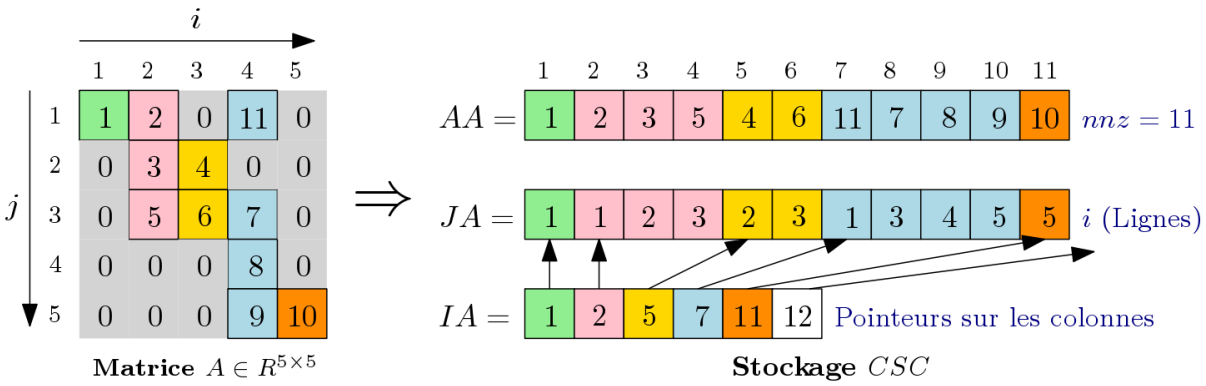

Figure 1.9: Stockage CSC de la matrice A.

1/ **Produit Matrice creuse A (Format de stockage COO) et vecteur x:**

Soit $A \in R^{m\times n}$ une matrice creuse et $x \in R^{n\times1}$. On cherche à calculer le produit $y = A \times x$. Donc, le vecteur résultat $y \in R^{m\times1}$.

L'algorithme dans listing (1.13) est basé sur le stockage COO d'une matrice creuse A.
Cet algorithme prend en compte les algorithmes de calcul des éléments non nuls de la matrice creuse A dans listing (1.8) et de stockage COO de la matrice creuse A dans listing (1.9).

Listing 1.13: Produit Matrice creuse Vecteur (Format COO)

```
%A partir du Stockage COO, on calcule: y=A*x

function [y]= COO_SPMatVec(A,x)

[m n]= size(A); %dimensions de la matrice A(m,n)
y=zeros(m,1); %initialisation du vecteur y

%Appel des fonctions NNZ_SPMat et COO_SPMat
[nnz]=NNZ_SPMat(A); %Fonction qui calcule nnz, le nombre des elements non nuls
    de la matrice A
[AA,JA,IA]=COO_SPMat(A); %Fonction qui calcule AA, JA, IA du stockage COO

for i=1:nnz
    y(JA(i)) = y(JA(i))+AA(i)*x(IA(i)); %Calcul du produit y=A*x
end

endfunction
funcprot(0)
```

(i) et (ii)/ L'algorithme de multiplication matrice creuse A vecteur x est testé et validé avec Scilab, comme montré dans listing (1.14). On considère deux cas de vecteurs x: $x = [1, 1, 1, 1, 1]'$ et $x_2 = [1, 0, 0, 1, 0, 0]'$.

Listing 1.14: Tests et Validation du produit matrice creuse vecteur

```
%Test de l'algo produit matrice creuse vecteur

%Test n1 (i)
A= [1,0,0,2,0;3,4,0,5,0;6,0,7,8,9;0,0,10,11,0;0,0,0,0,12]; %matrice A(5*5)
x = [1,1,1,1,1]'; %n=5 (Ex5,i)
[y]= COO_SPMatVec(A,x); %Algo produit Mat Vec
disp(y); %Display le produit de A*x

%Test de l'algo avec Scilab
y1=A*x; %produit de A et x
err=norm(y-y1); %erreur entre y de l'algo et y1 de Scilab
disp(err); %err=0 --> Algo valide

%Test n2 (ii)
A2= [5,0,0,22,0,-15;0,11,3,0,0,4;0,0,0,-6,0,0]; %matrice A(6*3)
x2 = [1,0,0,1,0,0]'; %n=6 (Ex5,ii)
[y2]= COO_SPMatVec(A2,x2); %Algo produit Mat Vec
disp(y2); %Display le produit de A2*x2

%Test de l'algo avec Scilab
y3=A2*x2; %produit de A2 et x2
err2=norm(y3-y2); %erreur entre y2 de l'algo et y3 de Scilab
disp(err2); %err=0 --> Algo valide

funcprot(0)
```

La complexité de l'algorithme de produit matrice creuse vecteur dans listing (1.13) est évaluée à O(nnz).

On peut calculer la moyenne de 10 mesures du temps de calcul de cet algorithme, comme indiqué dans le tableau (1.9).

Algorithme	**Moyenne du temps de calcul**
$y = A \times x$	0.0003155 s

Table 1.9: Moyenne de temps de calcul de l'algorithme Produit Matrice creuse Vecteur.

2/ Produit Matrice creuse A (Format de stockage CSR) et vecteur x:

Le produit matrice-vecteur en utilisant le format du stockage CSR est exprimé par la relation (1.10), *"CRS Matrix-Vector Product "* 2000:

$$y_i = \sum_j a_{i,j} \times x_j \tag{1.10}$$

L'algorithme dans listing (1.15) est basé sur le stockage CSR d'une matrice creuse A.

Cet algorithme tient en compte les algorithmes de calcul des éléments non nuls de la matrice creuse A dans listing (1.8) et de stockage CSR de la matrice creuse A dans listing (1.10).

Listing 1.15: Produit Matrice creuse Vecteur (Format CSR)

```
%A partir du stockage CSR, on calcule le produit: y=A*x

function [y]= CSR_SPMatVec(A,x)

[m n] = size(A); %Taille de la matrice A (m,n)
y = zeros(m,1); %initialisation du vecteur y de taille (m,1)
[AA,JA,IA]=CSR_SPMat(A); %appel a l'algo de stockage CSR

for i = 1:m-1
  for j=IA(i):IA(i+1)-1
    y(i) = y(i) + AA(j)*x(JA(j)); %vecteur y produit de la matrice creuse A et
    du vecteur x
  end
end
    y(m) = y(m) + AA(IA(m))*x(JA(m)); %valeur m du vecteur y
endfunction
funcprot(0)
```

Pour valider l'algorithme dans listing (1.15), on reprend l'algorithme du test dans listing (1.14) qui comprend le produit de 2 matrices vecteurs: $A \in R^{5\times5}$ et $x \in R^{5\times1} = [1,1,1,1,1]$, ainsi que $A_2 \in R^{6\times3}$ et $x_2 \in R^{6\times1} = [1,0,0,1,0,0]$.

Le stockage CSR de la première matrice creuse A est donné dans la figure (1.10).

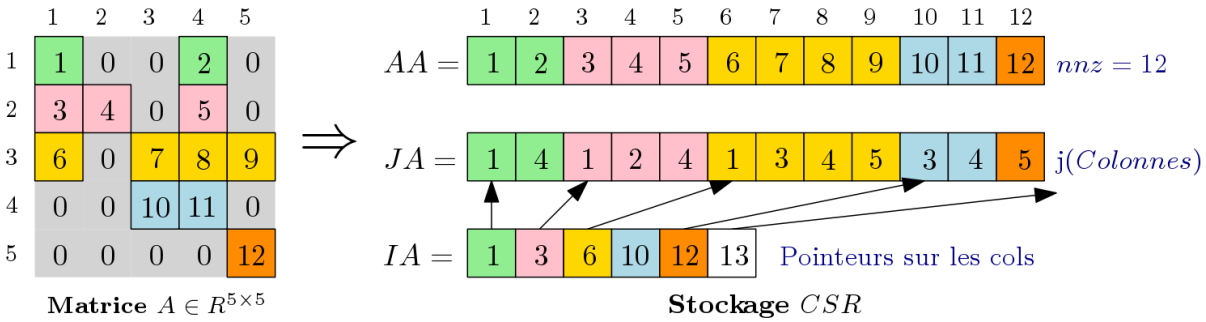

Figure 1.10: Stockage CSR de la matrice A.

Le produit de la matrice creuse A et du vecteur x est illustré dans la figure (1.11).

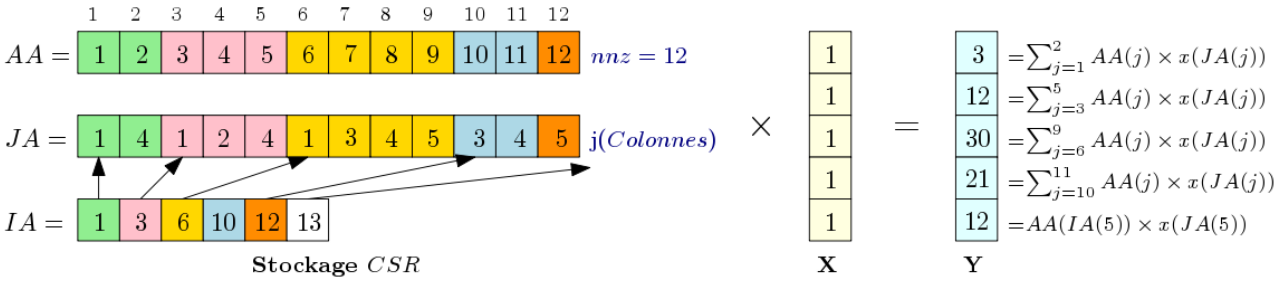

Figure 1.11: Produit de la matrice A vecteur x.

Ensuite, le temps de calcul du produit matrice creuse vecteur, format CSR est comparé à celui COO dans le tableau (1.10). Comme il s'agit de la même complexité, on peut remarquer que

le temps de calcul du produit matrice creuse vecteur pour les deux formats de stockage CSR et COO est presque le même.

Algorithme	**Mat-Vec COO**	**Mat-Vec CSR**
$y = A \times x$	0.0003155 s	0.000315s

Table 1.10: Moyenne de temps de calcul de l'algorithme Produit Matrice creuse Vecteur pour les formats de stockage COO et CSR.

En conclusion, la comparison entre les deux formats de stockage COO et CSR est donnée dans le tableau (1.11), *"Matrice Creuse"* 2020.

Stockage	**Le stockage sous forme COO**	**Le stockage sous forme CSR**
Avantages	- Facilité d'implémentation -Stockage pratique (sous forme de triplet (i,j,élément))	- Format compressé (IA est de taille $(n+1)$ au lieu de nnz) - Tableaux triés
Désavantages	Pour le produit matrice vecteur: deux adressages sont nécessaires	Couteux d'ajouter des éléments dans la matrice (il faut connaitre l'emplacement des éléments non nuls de la matrice creuse)

Table 1.11: Comparison entre les formats de stockage d'une matrice creuse: COO et CSR.

Chapter 2

TP5 de Calcul Numérique

2.1 Travail préliminaire: Cas de Test

Problématique:

On considère l'équation de la chaleur dans un milieu immobile linéaire homogène avec terme source et isotrope:

$$\begin{cases} -k\frac{\partial^2 T}{\partial x^2} &= g, \;\; x \in]0,1[\\ T(0) &= T_0 \\ T(1) &= T_1 \end{cases} \tag{2.1}$$

avec g est un terme source, $k > 0$ le coefficient de conductivité thermique et $T_0 < T_1$ les températures au bord du domaine considéré.

On se propose de résoudre cette équation par une méthode de différence finie centrée d'ordre 2. On discrétise le domaine $1D$ selon $(n+2)$ noeuds x_i, $i = 0, 1, ..., n+1$, espacés d'un pas h constant.

En chaque noeud, l'équation discrète s'écrit:

$$-k(\frac{\partial^2 T}{\partial x^2})_i = g_i \tag{2.2}$$

Exercice 1:

1/ Cherchons une écriture de la différence finie d'ordre 2 de la température T.

Soit le pas $h \in \mathbb{R}$. On suppose qu'il existe $\epsilon > 0$, qui vérifie:

$$|x + \epsilon| \leq h \tag{2.3}$$

D'après le développement de Taylor à l'ordre 2, on a:

$$\begin{cases} T(x+h) = T(x) + h\frac{\partial T}{\partial x} + \frac{h^2}{2}\frac{\partial^2 T}{\partial x^2} + O(h^2) \\ T(x-h) = T(x) - h\frac{\partial T}{\partial x} + \frac{h^2}{2}\frac{\partial^2 T}{\partial x^2} + O(h^2) \end{cases} \tag{2.4}$$

En sommant les deux égalités précédentes, on obtient:

$$T(x+h) + T(x-h) = 2T(x) + h^2\frac{\partial^2 T}{\partial x^2} + O(h^2) \tag{2.5}$$

Ainsi:

$$-\frac{\partial^2 T}{\partial x^2} = \frac{-T(x-h) + 2T(x) - T(x+h)}{h^2} \tag{2.6}$$

2/ On note pour $\forall i \in [0, n+1]$:

$$\begin{cases} T(x_i) = U_i \\ A \times U = f \end{cases} \tag{2.7}$$

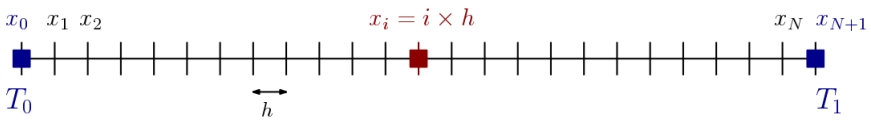

Figure 2.1: Interval de discrétisation (points de grille).

D'après la figure (2.1), le calcul de x_{N+1} nous permet de déduire le pas h:

$$x_{N+1} = (N+1) \times h = \frac{N+1}{N+1} = 1 \tag{2.8}$$

On déduit le pas h:

$$h = \frac{1}{N+1} \tag{2.9}$$

$$\begin{cases} T(x_0) = T_0 = U_0 \\ T(x_{N+1}) = T(1) = T_1 = U_{N+1} \end{cases} \tag{2.10}$$

D'après l'équation (2.6), le schéma centré d'ordre 2 est:

$$k\,\frac{-U_{i-1} + 2U_i - U_{i+1}}{h^2} = g_i, \quad \forall i \in [1, n] \tag{2.11}$$

Pour $i = 1$:

$$\begin{cases} k\,\frac{-U_2 + 2U_1 - U_0}{h^2} = g \\ U_0 = T_0 \end{cases} \tag{2.12}$$

En remplaçant U_0 par T_0 dans l'équation (2.12), on en déduit que:

$$2U_1 - U_2 = \frac{h^2}{k} g_1 + T_0 \tag{2.13}$$

Pour $i = 2$:

$$k \ \frac{-U_1 + 2U_2 - U_3}{h^2} = g_2 \tag{2.14}$$

On en déduit que:

$$-U_1 + 2U_2 - U_3 = \frac{h^2}{k} g_2 \tag{2.15}$$

Pour $i = N$:

$$\begin{cases} k \ \frac{-U_{N-1}+2U_N-U_{N+1}}{h^2} = g_N \\ T(x_{N+1}) = T_1 = U_{N+1} \end{cases} \tag{2.16}$$

On en déduit que:

$$-U_{N-1} + 2U_N = \frac{h^2}{k} g_N + T_1 \tag{2.17}$$

Ainsi, le système linéaire de dimension n correspondant au problème étudié:

$$\begin{cases} A \times U = f \\ A \in \mathbb{R}^{n \times n}, u \ \text{ et } \ f \in \mathbb{R}^n \end{cases} \tag{2.18}$$

D'où le système linéaire $AU = f$ suivant:

$$\underbrace{\begin{bmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & -1 & 2 & -1 & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \dots & 0 & -1 & 2 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} U_1 \\ U_2 \\ \vdots \\ U_{n-1} \\ U_n \end{bmatrix}}_{\mathbf{U}} = \underbrace{\begin{bmatrix} \frac{h^2}{k} g_1 + T_0 \\ \frac{h^2}{k} g_2 \\ \vdots \\ \frac{h^2}{k} g_{N-1} \\ \frac{h^2}{k} g_N + T_1 \end{bmatrix}}_{\mathbf{f}}$$

Dans la suite du TP, on considère qu' il n' y a pas de source de chaleur, i.e. $g = 0$.

Donc, la solution analytique de l'équation est:

$$T(x) = T_0 + x(T_1 - T_0) \tag{2.19}$$

2.2 Méthode directe et stockage bande

La matrice de poisson 1D étudiée est tridiagonale. Alors, on utilise le stockage par bande. Dans un premier temps, on va étudier le stockage de type General Band (GB), proposé lors de l'utilisation de LAPACK et BLAS.

Soit une matrice $A \in R^{m \times n}$. Elle comporte:

$$\begin{cases} kl \text{ sous-diagonales} \\ ku \text{ sur-diagonales} \end{cases}$$

Elle sera stockée d'une manière compacte dans un tableau $2D \in R^{(kl+ku+1)\times n}$ ($(kl + ku + 1)$ lignes et n colonnes), à condition que kl, $ku << \min(m, n)$. Ainsi la matrice est stockée comme suit:

- Les colonnes de la matrice sont stockées dans les colonnes correspondantes du tableau,
- Les diagonales de la matrice sont stockées dans les lignes du tableau.

Dans LAPACK, l'élément a_{ij} de la matrice A est stocké dans le tableau $AB(ku + 1 + i - j, j)$. A titre d'exemple, on illustre dans la figure (2.2) le cas d'exemple du TP5, qui schématise le stockage d'une matrice GB $A \in R^{5\times5}$ dans le tableau $2D$, AB.

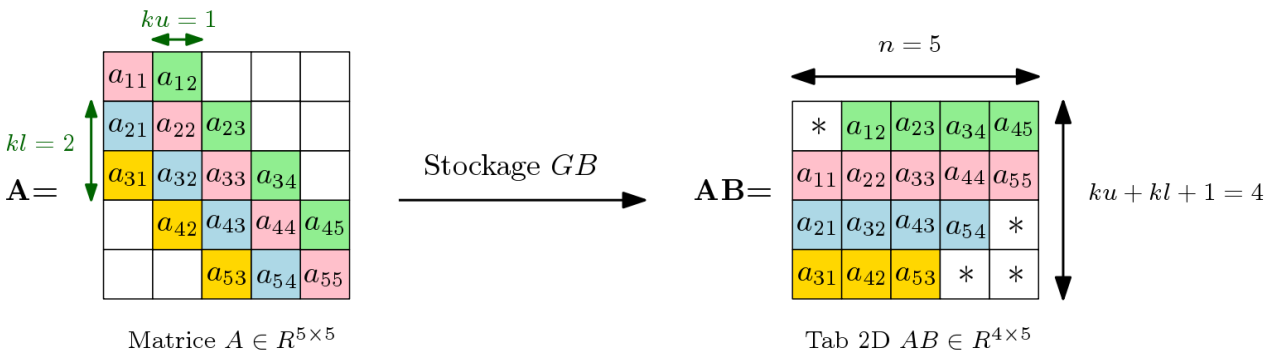

Figure 2.2: Cas d'exemple: stockage General Band de la matrice A.

avec les éléments $*$ généralement affectés par des 0.

Dans la suite du TP, on va étudier le code C associé au TP, en faisant appel à LAPACK et BLAS.

Exercice 2: Environnement C et BLAS/LAPACK

Préparation du travail et compilation du code source fournit:

1/ Il faut tout d'abord installer les dépendances nécessaires à la compilation du code C.

Sous Ubuntu, on utilise la commande: *sudo apt-get install libblas-dev liblapacke-dev.*

2/ **Compilation et exécution du code de test:**

- En tappant ***make*** dans la terminal, on obtient dans la répértoire du travail les exécutables (*.o*) suivants:

  lib_poisson1D.o, *tp2_poisson1D_direct.o*, *tp2_poisson1D_iter.o* et *tp_env.o*,

- En tappant ***make run_testenv*** dans la terminal, on obtient la sortie illustrée dans la figure (2.3).

```
bin/tp_testenv
--------- Test environment of execution for Practical exercises of Numerical Algorithmics --------
-

The exponantial value is e = 2.718282
The maximum single precision value from float.h is flt_max = 3.402823e+38
The maximum double precision value from float.h is dbl_max = 1.797693e+308
The epsilon in single precision value from float.h is flt_epsilon = 1.192093e-07
The epsilon in double precision value from float.h is dbl_epsilon = 2.220446e-16


 Test of BLAS/LAPACK environment using cblas_dcopy
x[0] = 1.000000, y[0] = 6.000000
x[1] = 2.000000, y[1] = 7.000000
x[2] = 3.000000, y[2] = 8.000000
x[3] = 4.000000, y[3] = 9.000000
x[4] = 5.000000, y[4] = 10.000000

Test DCOPY y <- x
y[0] = 1.000000
y[1] = 2.000000
y[2] = 3.000000
y[3] = 4.000000
y[4] = 5.000000


--------- End -----------
```

Figure 2.3: Exécution du code de test: *make run_testenv.*

Les bibliothèques utilisées dans le fichier d'en-tête "tp_env.h", *"tutorialspoint_C"* 2021:

- Bibliothèque $< math.h >$: définit les fonctions mathématiques,
- Bibliothèque $< float.h >$: définit les constantes liées aux valeurs à virgule flottante,
- Bibliothèque $< limits.h >$: limite les valeurs de divers types de variables,
- Les directives $< cblas.h >$ et $< lapacke.h >$ de LAPACK/ BLAS permettent d'appeler les fonctions de ces bibliothèques.

Les valeurs affichées dans la terminal sont:

- M_E (de la biblio. $< math.h >$): définit la constante e de type float, appelée constante de Néper (ou nombre d'Euler), évaluée à 2.718282, *"Macro* $M_E(NonStandard)$" 2021,
- FLT_MAX (de la biblio. $< float.h >$): définit la valeur maximale à virgule flottante. Elle est évaluée à $flt\_max = 3.402823e + 38$ (en écriture scientifique $\%e$),
- DBL_MAX (de la biblio. $< float.h >$): définit la valeur maximale double précision. Elle est évaluée à $dbl\_max = 1.797693e + 308$ (en écriture scientifique $\%e$),
- $FLT\_EPSILON$ (de la biblio. $< float.h >$): définit la différence entre 1 et la plus petite valeur représentable supérieure à 1 en simple précision. Elle est évaluée à $flt\_epsilon = 1.192093e - 07$ (en écriture scientifique $\%e$),
- $DBL\_EPSILON$ (de la biblio. $< float.h >$): définit la différence entre 1 et la plus petite valeur représentable supérieure à 1 en double précision. Elle est évaluée à $dbl\_epsilon = 2.220446e - 16$ (en écriture scientifique $\%e$).

Le test de l'environment BLAS/LAPACK est fait à l'aide de la fonction *cblas_dcopy*, qui permet de copier le vecteur $x \in R^{5\times1}$ dans un autre vecteur $y \in R^{5\times1}$.

3/ **Compilation et exécution du code qui résout l'équation de la chaleur $1D$ par une méthode directe:**

- En tappant ***make*** dans la terminal, on obtient dans la répértoire du travail les exécutables ($.o$).
- En tappant ***make run_tp2poisson1D_direct*** dans la terminal, on obtient la sortie illustrée dans la figure (2.4).

```
bin/tp2poisson1D_direct
--------- Poisson 1D ---------


 INFO DGBSV = 0

The relative residual error is relres = 5.889846e-15


--------- End -----------
```

Figure 2.4: Exécution du code de résolution de l'équation de la chaleur $1D$ par une méthode directe: *make run_tp2poisson1D_direct*.

Le code source *tp2_poisson1D_direct.c* contient la fonction principale de la résolution du problème de poisson $1D$ (Equation de chaleur). Tandis que l'en-tête *lib_poisson1D.h* contient la déclaration

des fonctions utiles pour la résolution de ce problème.

Dans *INFO DGBSV*, on affiche la valeur indicative *info* (int), qui indique la status de la sortie de la fonction *LAPACKE_dgbsv()*, dont les paramètres dépendent du layout de la matrice: *LAPACK_ROW_MAJOR* ou *LAPACK_COL_MAJOR*.

Selon son retour, l'entier *info* indique, *"DGBSV Subroutine"* 2019:

- si $info = 0$, alors *DGBSV* subroutine s'est terminé avec succès,
- si $info > 0$, alors *info* est égale à i pour lequel $U_{ii} = 0$ et on n'a pas de solution.

La fonction *LAPACKE_dgbsv()* est representée dans la figure (2.5). Ses paramètres seront détaillés par la suite.

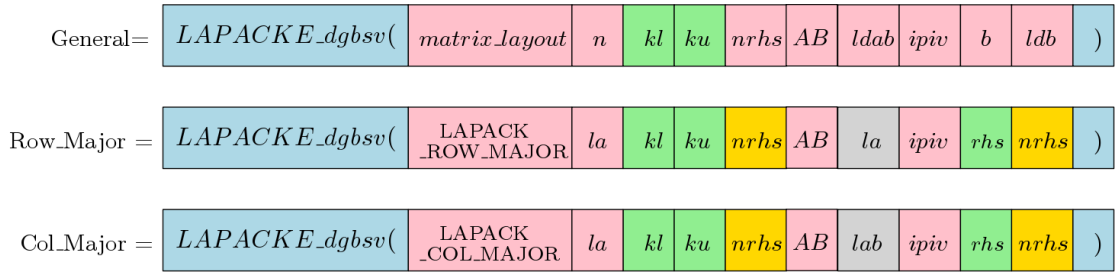

Figure 2.5: Fonction *LAPACKE_dgbsv()*.

On affiche aussi la valeur de l'erreur résiduelle relative *relres*.

Exercice 3: Utilisation de BLAS/LAPACK

Préambule:

Les bibliothèques standard BLAS (Basic Linear Algebra Subprograms) et LAPACK (Linear Algebra PACKage, pour effectuer un calcul spécifique) proposent des algorithmes (sous-programmes) de manipulations des vecteurs et matrices pour la résolution des équations linéaires, *"guide LAPACK"* 2019.

Les 3 niveaux présentés dans la bibliothèque BLAS:

- Niveau 1 pour les opérations vecteur-vecteur (VEC-VEC),
- Niveau 2 pour les opérations vecteur-matrice (MAT-VEC),
- Niveau 3 pour les opérations matrice-matrice (MAT-MAT).

Les nombres réels manipulés dans le code en C utilisant LAPACK/ BLAS sont en simple précision (32 bits) et double précision (64 bits).

1/ Le stockage d'une matrice A est basé sur différentes schémas utilises dans LAPACK, selon sa nature, " *Matrix Storage Schemes*" 1999. On peut citer le:

- stockage conventionnel (Full storage) dans un tableau 2D ($A[m][n]$),
- stockage emballé (Packed storage): matrices symétriques, hermitiennes ou triangulaires,
- stockage bande (Band storage): pour les matrices bande,
- stockage des matrices bidiagonales et tridiagonales dans 2 ou 3 tableaux 1D.

Dans notre cas, la matrice A de poisson 1D est une matrice bande tridiagonale (General Band). Ses diagonales (sous-diagonale, diagonale principale et super-diagonale) sont stockée dans un tableau AB (de doubles, passé en tant que pointeur) de deux dimensions $lab \times la$, avec $lab = kl + ku + 1 = 3$ et $la = nbpoints - 2 = 100$, comme illustré dans la figure (2.6) dans le cas de stockage ROW_Maj. Chaque argument du tableau 2D possède une dimension principale dans la liste des arguments (de la forme LD$<$ $nom - tableau$ $>$), "*Array Arguments*" 1999.

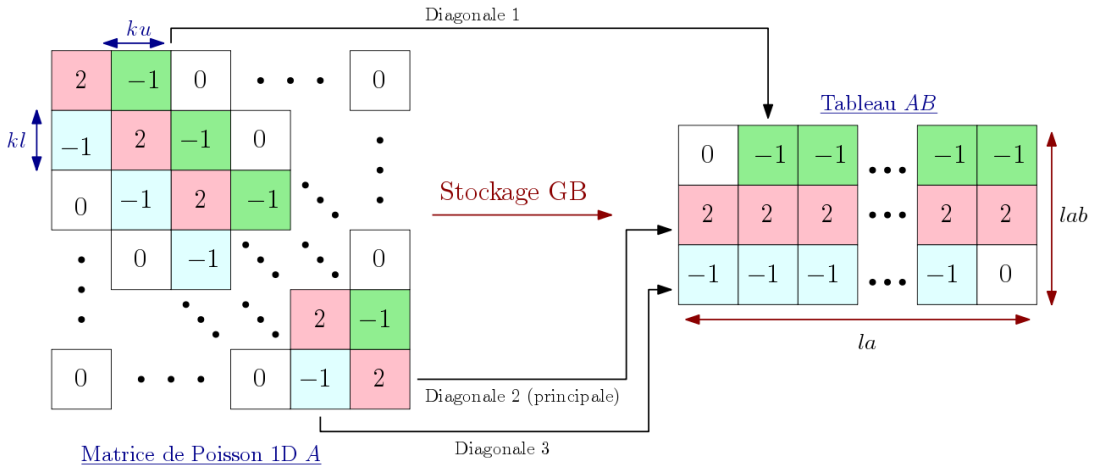

Figure 2.6: Stockage General Band de la matrice tridiagonale de Poisson 1D (en ROW_Maj).

En effet, il existe deux types de stockage des éléments d'une matrice A dans un tableau 2D AB:

- RowMajor (en C): les éléments contigus appartiennent à la même ligne,
- ColMajor (en Fortran): les éléments contigus appartiennent à la même colonne.

La gestion de la mémoire est assurée par *malloc* et *free*.

- L'allocation de mémoire pour le tableau AB est donné par la commande:
  $AB = (double*)malloc(sizeof(double) * lab * la)$,
- La mémoire est à la fin désallouée avec $free()$.

2/ **LAPACK_COL_MAJOR:** est définie dans lapacke.h (transposée de ROW_Major). Elle indique que le tableau est stocké dans l'ordre de colonne principale au lieu de la ligne principale, *"Array Arguments"* 1999.

Les tableaux 2D à colonnes principales (Col_Major) sont caractérisés par des :

- éléments de colonne contigus,
- éléments d'une colonne à l'autre séparés par la dimension principale.

3/ **La dimension principale** ld **(Leading Dimension):** dénote la longueur de la première dimension du tableau 2D. Elle indique la nature de stockage des éléments et dépend du layout de la matrice, *"The LAPACKE C Interface to LAPACK"* 2016:

- Représentation ROW_Maj: La dimension principale ld sépare les éléments d'une ligne à la suivante, ld sera dans ce cas le nombre des colonnes (Ici "la" d'après la figure (2.5)),
- Représentation COL_Maj: La dimension principale ld sépare les éléments d'une colonne à la suivante, ld sera dans ce cas le nombre des lignes (Ici "lab" d'après la figure (2.5)).

4/ **Fonction** *dgbsv*() **(General Band Scalar Vector of Double Precision):**

Elle calcule la solution d'un système réel d'équations linéaires $AX = B$, où A est une matrice bande d'ordre n et X et B des matrices générales de taille $n \times NRHS$, *"DGTSV Subroutine"* 2012 et *"DGBSV Subroutine"* 2019.

D'après la figure (2.5), on peut exprimer les paramètres de la fonction *dgbsv*:

- matrix_Layout (int): indique si les matrices sont stockées dans l'ordre principal des lignes (matrix_layout = LAPACK_ROW_MAJOR) ou l'ordre principal des colonnes (matrix_layout = LAPACK_COL_MAJOR),
- n (lapack_int): indique l'ordre de la matrice A et le nombre de lignes de la matrice B. Dans notre cas, elle est évaluée à "la",
- kl (lapack_int): indique la largeur de la bande inférieure (sous-diagonale),
- ku (lapack_int): indique la largeur de la bande supérieure (sur-diagonale),
- $NRHS$ (lapack_int, Number of Right-Hand Sides): indique le nombre de colonnes de la matrice B,

- AB (double *): indique le tableau 2D de taille $lab \times la$ dans lequel on a stocké la matrice bande,
- $ldab$ (lapack_int, Leading Dimension of the Array AB): indique la dimension principale de AB, égale à "la" dans le cas où "matrix_layout = LAPACK_ROW_MAJOR" et à "lab" dans le cas où "matrix_layout = LAPACK_COL_MAJOR",
- $ipiv$ (lapack_int *): indique les éléments contenants les indices du pivot. Il s'agit d'un vecteur de longeur "la",
- b (double *): indique la matrice générale B, contenant les $nrhs$ membres droits du système. Dans notre cas, elle est évaluée à $nrhs$,
- ldb (lapack_int): indique la dimension principale de B. Dans notre cas, elle est évaluée à $nrhs$.

La fonction *dgbsv* implémente la méthode de factorisation LU avec pivot partiel et échange de lignes. Ainsi, la matrice A se factorise comme étant un produit de deux matrices triangulaires inférieure L et supérieure U avec, *"DGBSV Documentation"* 2006:

- L est le produit de permutation et de matrice triangulaires inférieures unitaires avec kl sous-diagonales,
- U est une matrice triangulaire supérieure avec $(kl + ku)$ super-diagonales.

Finalement, la forme factorisée $A = LU$ est utilisé pour résoudre le système d'équations $AX = B$.

5/ **Stockage en priorité ligne:** On modifie *tp2_poisson1D_direct.c* et *lib_poisson1D.c*, pour effectuer les opérations pour un stockage en priorité ligne, au lieu du stockage colonne.

Formats de stockage: ROW_Major et COL_Major

On pose $kv = 0$, puis on exécute le programme dans le cas de COL_Major et ROW_Major. Dans ce cas, on a: $lab = ku + kl + 1$.

La figure (2.7) présente les deux configurations de stockage de la matrice tridiagonale de poisson 1D obtenue dans les fichiers de sortie *AB_col.dat* et *AB_row.dat*, tel que:

- Dans le cas de ROW_Major (a), on obtient un tableau 2D de taille $lab \times la = 3 \times 100$,
- Dans le cas de COL_Major (b), on obtient un tableau 2D de taille $la \times lab = 100 \times 3$.

avec $n = 100$ représentant le *nombre de points* $- 2$, et kl et ku représentant les sous- et sur-diagonales respectivement.

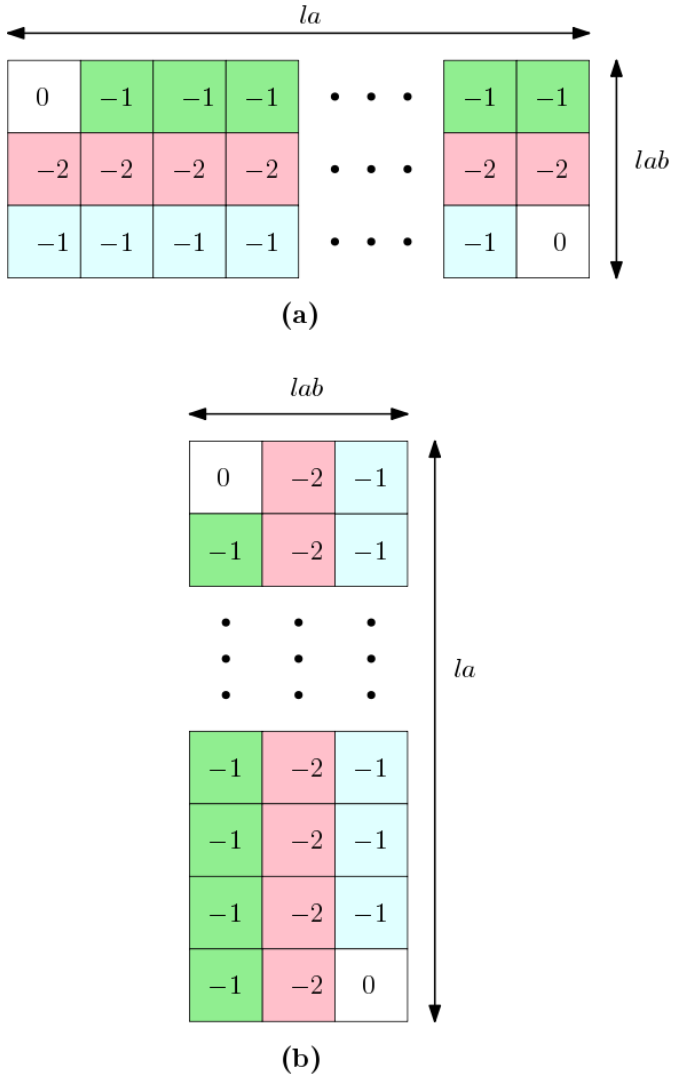

Figure 2.7: Formes de stockage de la matrice tridiagonale de Poisson 1D. (a) Stockage en Row Major, (b) stockage en Col Major.

Exercice 4: DGBMV

0/ ***cblas_dgbmv* (General Band Matrix-Vector operation of double:)**

D'après *"DGBMV Subroutine"* 2021, la fonction *dgbmv* permet de réaliser une des opérations matrice-vecteur suivantes:

$$y := \alpha Ax + \beta y \quad \text{ou} \quad y := \alpha A^T x + \beta y \tag{2.20}$$

avec: α et β des scalaires, x et y des vecteurs et A une matrice bande de dimension $m \times n$, possédant kl sous-diagonales et ku super-diagonales.

La figure (2.8) illustre les paramètres de la fonction *dgbmv*.

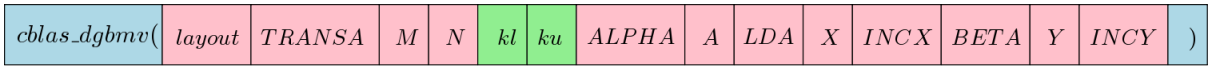

Figure 2.8: Paramètres de la fonction *cblas_dgbmv()*.

Ainsi, les paramètres sont décrites comme suit:

- *layout* (CBLAS_LAYOUT): indique le format de stockage (Row_Major ou Col_Major).
  - Si le stockage est en priorité ligne, alors $layout = CblasRowMajor$,
  - Si le stockage est en priorité colonne, alors $layout = CblasColMajor$.
- $TRANS$ (character): indique l'opération à effectuer:
  - Pour $y := alphaAx + betay$, on a $TRANSA = N$,
  - Pour $y := alphaA^T x + betay$, on a $TRANSA = T$ ou $TRANSA = C$.
- M (const int): indique le nombre de lignes de la matrice A,
- N (const int): indique le nombre de colonnes de la matrice A,
- kl (const int): indique le nombre de sous-diagonales de la matrice A,
- ku (const int): indique le nombre de sur-diagonales de la matrice A,
- $ALPHA$ (const double): indique le scalaire que multiplie la matrice A,
- A (const double*, pointeur): indique la matrice A,
- LDA (const int): indique la dimension principale du tableau contenant la matrice A. (au moins $kl + ku + 1$.)
- X (const double*, pointeur): indique le vecteur X,
- $INCX$ (const int): indique le "*stride within* X",
- $BETA$ (const double): indique le scalaire que multiplie le veteur Y,
- Y (double*, pointeur): indique le vecteur Y,
- $INCY$ (const int): indique le "*stride within* Y".

avec: *Stride d'un tableau (ou increment)* est le nombre d'emplacement mémoire entre les débuts d'éléments successifs du tableau (mesuré en bytes), *"Stride of an array"* 2020.

1/ **Fonction BLAS *dgbmv* pour le stockage ligne et colonne:**

La fonction *dgbmv* exécute l'opération donnée par l'équation (2.20). Dans notre cas, on a le système $AU = f$. Alors,on considère que:

$$\begin{cases} \alpha = 1 \\ \beta = 0 \end{cases} \tag{2.21}$$

Le système linéaire de l'équation (2.20) devient:

$$Y := AX \tag{2.22}$$

2/ **Validation de BLAS *dgbmv*:**

On peut valider la fonction *dgbmv()* en calculant l'erreur relative suivante:

$$relres = \frac{\|SOL\_Exacte - Y\|}{\|Y\|} \tag{2.23}$$

Après compilation, l'erreur obtenue est dans ce cas évaluée à: $6.404746e-16$, ce qui est faible et au périphérie des erreurs de l'ordinateur.

Suite aux tests éffectués, on constate que *dgbmv* ne marche pas dans le cas de la configuration Row_Major. Ceci est dû au fait que la fonction *dgbmv* des bibliothèques BLAS/ LAPACK ne supporte pas le stockage ligne principale pour la language C.

Exercice 5: LU pour les matrices tridiagonales

0/ **Pourquoi kv ne doit pas être Zéro?**

Dans le cas où on utilise les fonctions de BLAS/ LAPACK dans le calcul de la factorisation LU de la matrice bande A:

Lors de l'utilisant de *dgbtrs* dans le calcul de LU, pour la factorisation d'une matrice bande A (stockée dans un tableau 2D AB) en utilisant l'élimination de Gauss avec pivot partiel, une erreur relative aux arguments d'entrée est $2kl + ku + 1 > ldab$ (leading dimension). Dans notre cas, si $ldab = kl + ku + 1$ (cas où $kv = 0$): on aura $ldab = 3$ et $2kl + ku + 1 = 4$. Ce qui conduit au fait que $2kl + ku + 1 > ldab$. D'où l'erreur, *"General Band Matrix Factorization"* 2018.

1/ **Implémentation de la méthode de factorisation LU pour les matrices tridiagonales:**
On se propose de résoudre le système d'équations linéaires: $AX = B$, où A est matrice tridiagonale bande, stockée dans un tableau 2D $AB \in R^{la \times lab}$ avec $lab = 3$ (Priorité colonne) et X et B deux vecteurs de tailles la. Les étapes de résolution de cette équation par la méthode de factorisation LU de la matrice tridiagonale A est la suivante:

$$\begin{cases} LY = B \ \text{(Méthode de Descente)} \\ UX = Y \ \text{(Méthode de Remontée)} \end{cases} \tag{2.24}$$

Suite à la factorisation LU de la matrice A, le système devient:

$$LUX = B \tag{2.25}$$

Le code en C de la factorisation LU du tableau AB, dans lequel la matrice A est stockée est implémenté dans le fichier *tp_Lu.c*. Il comporte les fonctions suivantes:

- Factorisation LU de la matrice A,
- Méthode de descente $LY = B$ pour chercher Y,
- Méthode de remontée $UX = Y$ pour chercher X.

2/ **Validation de la factorisation LU:**
On peut valider le code en C de la factorisation LU de la matrice A stockée dans le tableau 2D AB en calculant l'erreur relative donnée dans l'equation (2.26), où EX_SOL est la solution exacte et B est la solution calculée.

$$relres = \frac{\|EX\_SOL - B\|}{\|B\|} \tag{2.26}$$

En compilant le code, on obtient une erreur équivalente à: $relres = 6.196127e-15$, donc proche de l'erreur machine.

2.3 Méthode de résolution itérative

Préambule:

Dans cette section, on s'intéresse a la résolution de l'équation de chaleur par une méthode itérative. Dans un premier temps, on effectue une étude theorique puis un maquettage avec Scilab. Dans un second temps, on implémente le code en C.

Excercice 6: Etude des méthodes Jacobi et Gauss-Seidel

1, 4/ **Méthodes de Jaccobi et Gauss-Seidel:**

Jaccobi et Gauss Seidel sont deux méthodes itératives pour la résolution de systèmes d'équations linéaires $Ax = b$ à une dimension finie, avec $A \in R^{n,n}$ une matrice carrée et x et $b \in R^{n\times 1}$.

On se base sur la forme générale dans l'implémentation des algorithmes de Gauss Seidel et Jaccobi. Soit la forme de découpage de la matrice A donnée par l'équation ci-dessous.

$$A = D - E - F \tag{2.27}$$

avec:

$$\begin{cases} D = diag(A) \ : \text{Matrice diagonale} \\ E \ : \text{Matrice triangulaire inférieure sans diagonale} \\ F \ : \text{Matrice triangulaire supérieure sans diagonale} \end{cases} \tag{2.28}$$

La figure (2.9) illustre le découpage de la matrice A en D, $-E$ et $-F$, comme indiqué précédemment dans les equations (2.27) et (2.28).

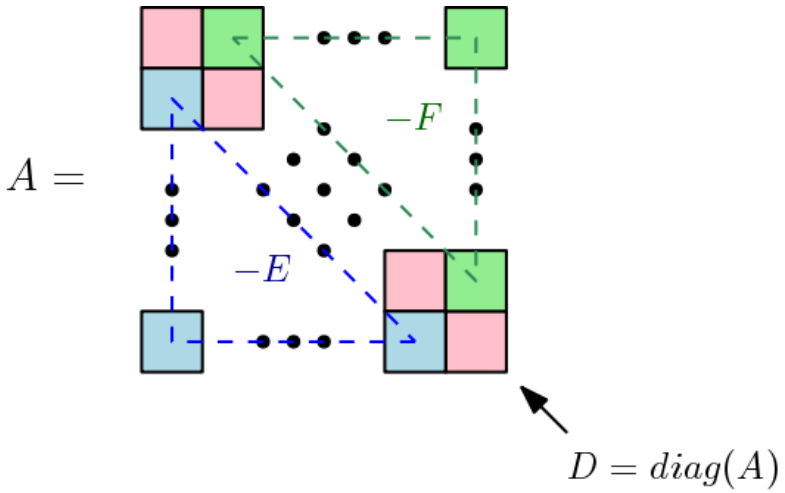

Figure 2.9: Découpage de la matrice A.

D'après le cours de ***Calcul Numérique***, la formule générale de Richardson s'ecrit:

$$x^{k+1} = x^k + M^{-1}(b - Ax^k) \text{ tant que } \|r^{k+1}\| > \epsilon \tag{2.29}$$

où:

- k est l'itération (doit être inférieur à un nombre maximum d'itérations),
- r est le vecteur résidu qu'on cherche à annuler ses composantes. Soit:

$$r = b - Ax \tag{2.30}$$

- M est une matrice inversible. Sa forme diffère en fonction de la méthode utilisée, comme suit:

$$\begin{cases} M = D \text{ (Méthode de Jaccobi)} \\ M = D - E \text{ (Méthode de Gauss Seidel)} \end{cases} \tag{2.31}$$

Les codes des méthodes itératives Jaccobi et Gauss-Seidel, décrites ci-dessus, sont implementés dans Scilab comme donné dans listings (2.1) et (2.2).

Listing 2.1: Algorithme de Jaccobi

```
function [Xk,r,relres,iter]=Jaccobi(A,b, k_max, epsilon)

    n = size(A,1); %taille de A
    Xk = zeros(n,1); %initialisation de Xk

    D= inv(diag(diag(A))); %matrice D diagonale

    iter=0; %compteur
    r = b - A * Xk; %initialisation residu
    relres = norm(r)/norm(b); %erreur residuelle

    while (relres>epsilon) && (iter< k_max) do
        Xk = Xk + D * r; %Richardson avec M=D
        r = b - A * Xk; %Residu
        iter = iter+1; %incrementation du compteur
        relres = norm(r) / norm(b); %erreur residuelle
    end
endfunction
funcprot(0)
```

Listing 2.2: Algorithme de Gauss Seidel

```
function [Xk,r,relres,iter]=gauss_Seidel(A,b, k_max, epsilon)

    n = size(A,1); %taille de A
    E = zeros (n,n); %initialization de E
    Xk = zeros(n,1); %initialisation de Xk

    D= diag(diag(A)); %matrice D diagonale
    E=-tril(A,-1); %matrice E
    M = inv(D-E); %M = inv(D-E)

    iter=0; %compteur
    r =  b - A * Xk; %initialisation residu
    relres= norm(r)/norm(b); %erreur residuelle

    while (relres>epsilon) && (iter< k_max) do
        Xk = Xk + M * r; %Richardson
        r = b - A * Xk; %Residu
        iter = iter+1; %incrementation du compteur
        relres = norm(r)/norm(b); %erreur residuelle
    end

endfunction
funcprot(0)
```

2/ **Analyse de complexité:** La complexité de Jaccobi évolue en O(n^2), tandis que celle de Gauss Seidel évolue en O($2n^2$). Ainsi, une itération de Gauss Seidel coûte deux fois plus cher que celle de Jaccobi (Gauss Seidel atteint la convergence deux fois plus vite que Jaccobi).
On peut améliorer les algorithmes en utilisant le stockage de la matrice tridiagonale bande A dans un tableau 2D AB. De plus, on peut faire appel aux fonction *dgbsv* de BLAS/ LAPACK pour la résolution des systemes linéaires $Ax = b$.

3, 5/ On pose $n = 3$. Le découpage de la matrice A de poisson 1D en des matrices D, E et F est illustré dans la figure (2.10).

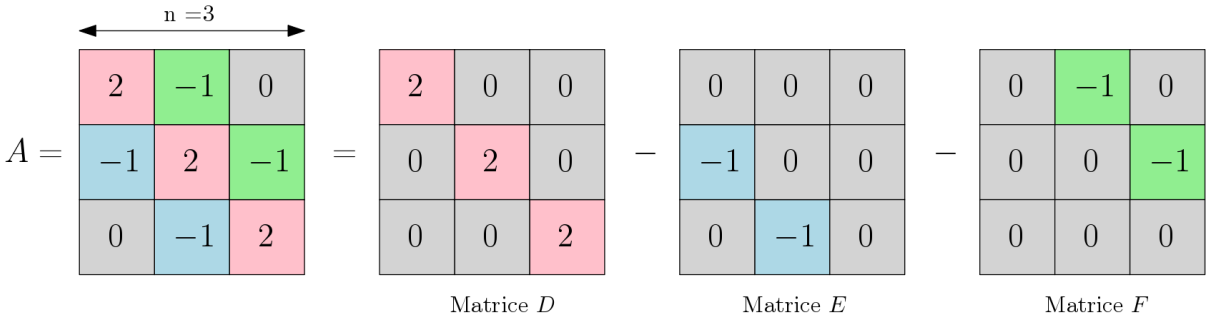

Figure 2.10: Découpage de la matrice A de poisson 1D, $n = 3$.

On exécute les deux algorithmes dans listing (2.1) et (2.2) avec ajout du nombre d'itérations *iter* comme paramètre de sortie. Les valeurs de nombre d'itérations en fonction de ϵ est présenté dans le tableau (2.1).

ϵ	**Jaccobi**	**Gauss Seidel**
10^{-1}	0	0
10^{-2}	9	2
10^{-3}	17	6
10^{-4}	25	9
10^{-5}	33	12
10^{-6}	41	16
10^{-7}	49	19
10^{-8}	57	22
10^{-9}	65	26

Table 2.1: Evolution du nombre d'itérations de Jaccobi et Gauss Seidel en fonction de ϵ.

La variation de nombre d'itérations en fonction de ϵ est illustré dans la figure (2.11). On constate qu'il faut deux fois plus d'iterations pour Jaccobi pour atteindre la convergence que Gauss Seidel. Ce qui prouve que ce dernier atteind la convergence deux fois plus vite que Jaccobi.

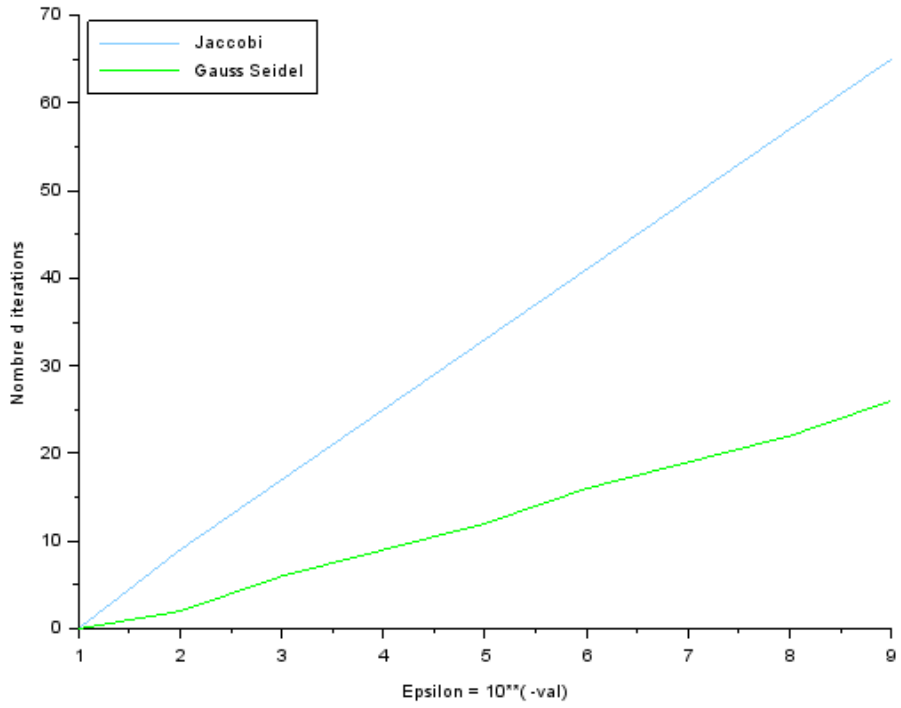

Figure 2.11: Comparison de nombre d'itérations jusqu'au convergence en fonction de ϵ.

Dans un second temps, on fixe ϵ à 10^{-9}, puis on calcule le nombre d'itérations pour atteindre la convergence en variant la taille n de la matrice A, comme montré dans le tableau (2.2).

	Jaccobi	**Gauss Seidel**
10	403	182
20	1397	600
30	2937	1218
40	4999	2014
50	7565	2976
60	10621	4091
70	14158	5351
80	18166	6748
90	22637	8275
100	27563	9926

Table 2.2: Evolution du nombre d'itérations de Jaccobi et Gauss Seidel en fonction de n, cas $\epsilon = 10^{-9}$.

La variation du nombre d'itérations en fonction de la taille de matrice n est illustrée dans la figure (2.12). On peut tirer comme conclusion que le nombre d'itérations évolue d'une manière quadratique en fonction de n, ce qui confirme la complexité des algorithmes de Jaccobi et Gauss Seidel.

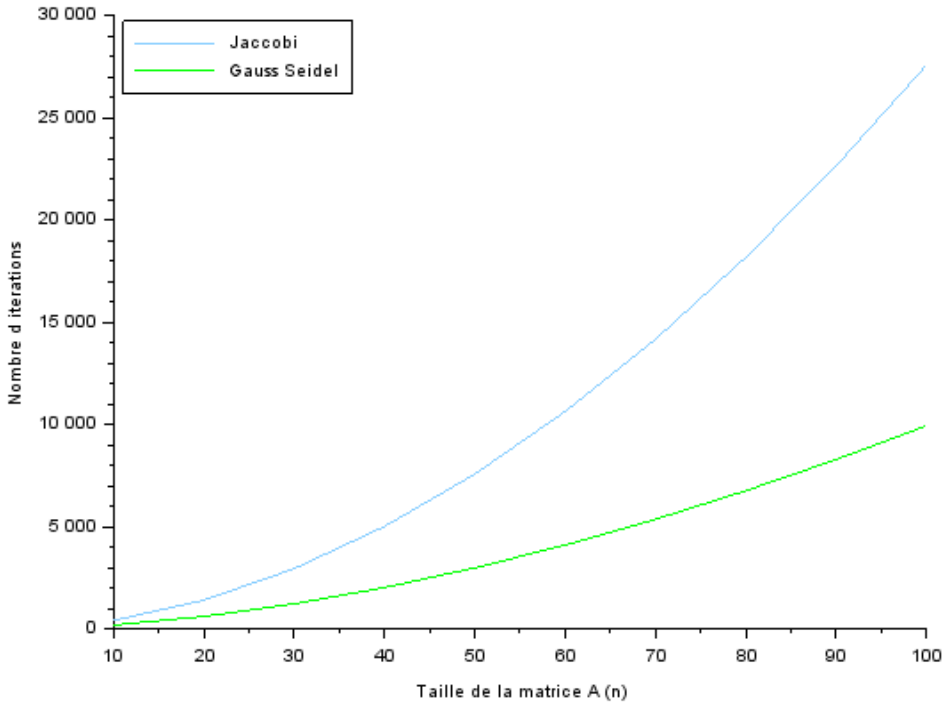

Figure 2.12: Comparison de nombre d'itérations en fonction de n, cas $\epsilon = 10^{-9}$.

Exercice 7: Etude du processus Itératif de Richardson:

Soit l'itération de Richardson:

$$x^{k+1} = x^k + \alpha(b - Ax^k), \quad \alpha \in \mathbb{R}^{*+} \tag{2.32}$$

1/ **Matrice d'itération G_α:** Soit la matrice d'itération G_α définie par:

$$x^{k+1} = G_\alpha x^k + \alpha b \tag{2.33}$$

D'après l'équation (2.32), on a:

$$x^{k+1} = Ix^k + \alpha b - \alpha Ax^k = (I - \alpha A)x^k + \alpha b \tag{2.34}$$

En comparant les équations (2.33) et (2.34), on obtient:

$$G_\alpha = I - \alpha A \tag{2.35}$$

2/ **Encadrement pour les valeurs propres de G_α:** Soit $A \in \mathbb{R}^{n\times n}$ une matrice carrée. Alors, A admet n valeurs propres λ_i, $i \in [1, n]$. On pose:

$$\begin{cases} \lambda_{max} = max(\lambda_i) \\ \lambda_{min} = min(\lambda_i) \end{cases} \tag{2.36}$$

D'après l'équation (2.35), on conclut que G_α admet aussi n valeurs propres γ_i. D'où l'encadrement:

$$1 - \alpha\lambda_{min} < \gamma_k < 1 - \alpha\lambda_{max}, \quad k \in [1, n] \tag{2.37}$$

3/ **Conditions sur α pour que la méthode converge:** On étudie le signe des valeurs propres de la matrice A. Alors, deux cas se présentent.

- Cas 1: $\lambda_{min} < 0$ et $\lambda_{max} > 0$: Ceci nous donne que $1 - \alpha\lambda_{max} < 1$ et $1 - \alpha\lambda_{min} > 1$

  Donc, il existe au moins une valeur propre λ_k de $\gamma_k > 1$.

  Le rayon spectral $\rho(G_\alpha) > 1$. Donc pas de convergence.

- Cas 2: $\lambda_{min} > 0$: Alors toutes les valeurs propres de A sont positives. On aura convergence si $\rho(G_\alpha) < 1$. D'où l'encadrement de α:

$$0 < \alpha < \frac{2}{\lambda_{max}} \tag{2.38}$$

4/ **Valeur Optimale de** α : On cherche à trouver $\alpha_{optimal}$ tel que: $\rho(G_{\alpha_{optimal}})$ soit minimale. Le rayon sepctral $\rho(G_\alpha)$ est donné par l'équation suivante:

$$\begin{aligned} \rho(G_\alpha) &= max\{|1 - \alpha\Lambda_{min}|, |1 - \alpha\Lambda_{max}|\} \\ \Rightarrow \rho(G_\alpha) &= \{(1 - \alpha\Lambda_{min}), (-1 + \alpha\Lambda_{max})\} \end{aligned} \tag{2.39}$$

En faisant l'égalité entre les deux termes de $\rho(G_\alpha)$, on obtient la valeur optimal de α:

$$\alpha_{optimal} = \frac{2}{\lambda_{min} + \lambda_{max}} \tag{2.40}$$

5/ **Implémentation Scilab:** On utilise dans l'implémentation de la méthode de Richardson dans Scilab l'équation (2.32). Soit l'algorithme presenté dans listing (2.3).

Listing 2.3: Algorithme de Richardson

```
function [Xk,r,relres,iter]=richardson(A,b, k_max, epsilon, alpha)

    n = size(A,1); %taille de A
    Xk = zeros(n,1); %initialisation de Xk

    iter=0; %Initialisation du compteur
    r =  b - A * Xk; %initialisation residu
    relres= norm(r)/norm(b); %erreur residuelle

    while (relres>epsilon) && (iter< k_max) do
        Xk = Xk + alpha * r; %Richardson, eq(2.32)
        r = b - A * Xk; %Residu r
        iter = iter+1; %incrementation du compteur
        relres = norm(r)/norm(b); %erreur residuelle
    end

endfunction
funcprot(0)
```

6/ **Convergence:** On propose dans listing (2.4) l'algorithme de test et validation pour les méthodes itératives: Jaccobi, Gauss Seidel et Richardson.

Listing 2.4: Test et validation des méthodes itératives label

```
%Calcul de nombre d iterations de Jaccobi, Gauss Seidel et Richardson

n = 3; %size of the matrix A
T0 = -5; %left boundary condition: T0
T1 = 5; %Right boundary condition: T1
k_max = 10D5; %max iterations
epsilon = 10D-2; %epsilon

[b] = RHS_vec(n, T0, T1); %vecteur b
[A] = Poisson1DMatric(n); %matrice de poisson 1D

lambda_max = max(spec(A)); %max de val propre (A)
lambda_min = min(spec(A)); %min de val propre (A)
alpha = 2/(lambda_max + lambda_min); //calcul de alpha optimale

[Xk,r,relres,iter]=Jaccobi(A,b, k_max, epsilon); % Methode de Jaccobi
disp(iter); %afficher le nombre d iterations pour Jaccobi

[Xk,r,relres,iter]=gauss_Seidel(A,b, k_max, epsilon); % Methode de Gauss Seidel
disp(iter); %afficher le nombre d iterations pour Gauss Seidel

[Xk,r,relres,iter]=richardson(A,b, k_max, epsilon, alpha); % Methode de
    Richardson
disp(iter); %afficher le nombre d iterations pour Richardson
```

Dans un premier temps, on compare le nombre des itérations des trois méthodes itératives étudiées en fonction de n pour $\alpha = \alpha_{optimal}$ et $\epsilon = 10^{-9}$, comme donné dans le tableau (2.3).

n	**Jaccobi**	**Gauss Seidel**	**Richardson**
10	403	182	403
20	1397	600	1397
30	2937	1218	2937
40	4999	2014	4999
50	7565	2976	7565
60	10621	4091	10621
70	14158	5351	14158
80	18166	6748	18166
90	22637	8275	22637
100	27563	9926	27563

Table 2.3: Evolution du nombre d'itérations de Jaccobi et Gauss Seidel en fonction de n, cas $\epsilon = 10^{-9}$.

On remarque que la méthode de richardson évolue avec le même nombre d'itérations que la méthode de Jaccobi. En effet, dans le cas de Jaccobi, on a $D(i,i) = \frac{1}{2}$ qui est un scalaire.
La méthode de Jaccobi est alors un cas particulier de celle de Richardson.

Dans un deuxieme temps, on se propose d'étudier le nombre d'itérations pour différentes valeurs de α.

Valeurs possibles de α pour que la méthode de Richardson converge:

Pour $n = 10$, on cherche à l'aide de la fonction *max(spec(A))* de Scilab la valeur propre maximale de la matrice A. Elle est évaluée à 0.5103361. D'après l'equation (2.38), on doit choisir α vérifiant: $0 < \alpha < \frac{2}{0.5103361} \approx 3.918985939$, pour que la méthode de Richardson soit convergente.

α	**Nombre d'itérations**
0.2	264
0.5	403
0.8	938
1	665
1.2	544
1.5	450
1.8	396
2	371
2.5	328
3	300
3.5	280
3.6	277
3.7	274
3.8	271
3.9	268
3.918985939	268

Table 2.4: Evolution du nombre d'itérations de Richardson en fonction de α, cas $\epsilon = 10^{-9}$ et $n = 10$.

On trace la courbe de l'évolution du nombre d'itérations de Richardson en fonction de α dans le cas d'étude ($\epsilon = 10^{-9}$ et $n = 10$).

Calcul de $\alpha_{optimal}$: Pour $n = 10$ et à l'aide des fonctions *max(spec(A))* et *min(spec(A))* de Scilab, on calcul les valeurs max et min des valeurs propres de A. On obtient:

$$\begin{cases} max(spec(A)) = 3.9189859 \\ min(spec(A)) = 0.0810141 \end{cases} \tag{2.41}$$

D'après les équations (2.40) et (2.41), on peut calculer $\alpha_{optimal}$:

$$\alpha_{optimal} = \frac{2}{0.0810141 + 3.9189859} = 0.5 \tag{2.42}$$

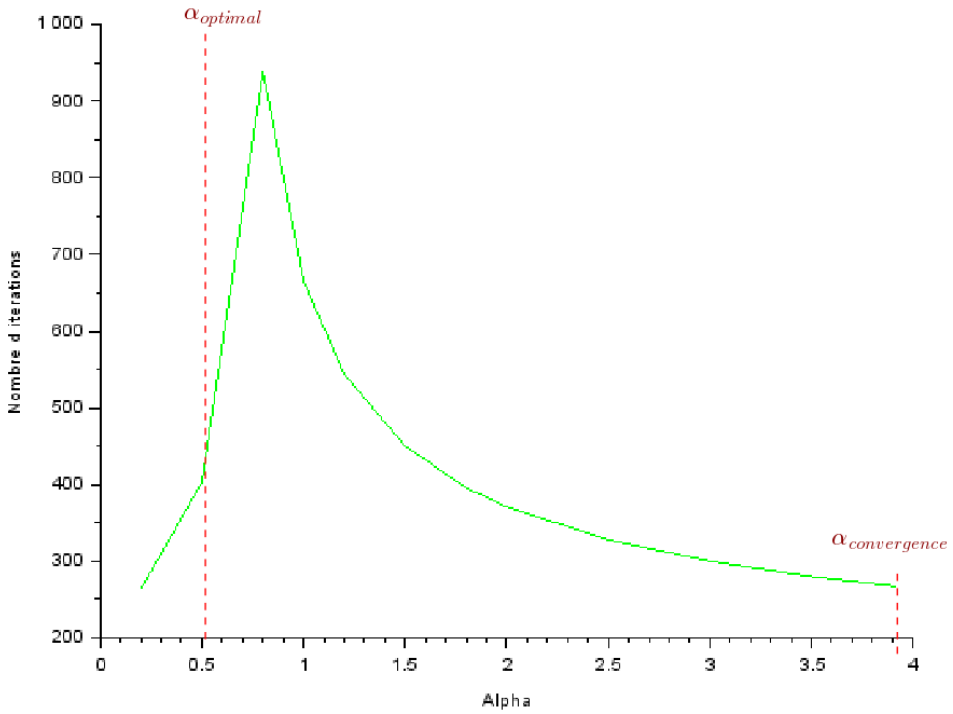

Figure 2.13: Nombre d'itérations de Richardson en fonction de α, cas $\epsilon = 10^{-9}$ et $n = 10$.

La figure (2.13) montre la convergence de la méthode de Richardson pour des valeurs de $\alpha \in [0, \alpha_{convergence}]$. Donc l'étude théorique est bien verifiée.

References

" *Matrix Storage Schemes*" (1999). netlib. URL: https://www.netlib.org/lapack/lug/node121.html.

"Algèbre Linéaire Numérique" (2006). Universite de Rennes. URL: https://perso.univ-rennes1.fr/eric.darrigrand-lacarrieu/Teaching/PdfFiles/PolyF04cours.pdf.

"Array Arguments" (1999). netlib. URL: http://www.netlib.org/lapack/lug/node116.html.

"Arrondis des ordinateurs, erreurs de mesure" (2021). URL: http://serge.mehl.free.fr/anx/arrondis_math.html.

"Chapitre 3 Systèmes d'équations" (2015). U. Laval Dept. Math Stat MAT. URL: https://www2.mat.ulaval.ca/fileadmin/Cours/MAT-2910/H-14/semaine6-new.pdf.

"CRS Matrix-Vector Product " (2000). Netlib. URL: http://www.netlib.org/utk/people/JackDongarra/etemplates/node382.html.

"DGBMV Subroutine" (2021). netlib. URL: http://www.netlib.org/lapack/explore-html/d7/d15/group__double__blas__level2_ga0dc187c15a47772440defe879d034888.html#ga0dc187c15a47772440defe879d034888.

"DGBSV Documentation" (2006). netlib. URL: http://www.netlib.org/lapack/lapack-3.1.1/html/dgbsv.f.html.

"DGBSV Subroutine" (2019). IBM. URL: https://www.ibm.com/docs/en/essl/6.2?topic=blaes-sgbsv-dgbsv-cgbsv-zgbsv-general-band-matrix-factorization-multiple-right-hand-side-solve.

"DGTSV Subroutine" (2012). LAPACKE. URL: http://www.netlib.org/lapack/explore-html-3.4.2/d4/d62/group__double_g_tsolve.html#.

"General Band Matrix Factorization" (2018). IBM. URL: https://www.ibm.com/docs/en/essl/6.1?topic=blaes-sgbtrf-dgbtrf-cgbtrf-zgbtrf-general-band-matrix-factorization.

"guide LAPACK" (2019). Jean Hare, Jacques Lefrère. URL: http://wwwens.aero.jussieu.fr/lefrere/master/mni/mncs/guide-lapack.pdf.

"Loi d'inertie de Sylvester" (2021). wikipedia. URL: https://fr.wikipedia.org/wiki/Loi_d%27inertie_de_Sylvester.

"Macro $M_E(NonStandard)$" (2021). koor.fr. URL: https://koor.fr/C/cmath/M_E.wp.

"Matrice Creuse" (2020). URL: https://bthierry.pages.math.cnrs.fr/course-fem/lecture/elements-finis-triangulaires/matrice-creuse/.

"Stride of an array" (2020). Wikipedia. URL: https://en.wikipedia.org/wiki/Stride_of_an_array.

"The LAPACKE C Interface to LAPACK" (2016). The LAPACKE C Interface to LAPACK. URL: https://www.netlib.org/lapack/lapacke.html.

"tutorialspoint_C" (2021). tutorialspoint. URL: https://www.tutorialspoint.com/index.htm.

Appendix A

Appendix Chapter

Les TPs 4 et 5 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