
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,
- Comparison de $LDL^T$ à $LU$,
- 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 le format COO et CSR,
- Test et validation, étude de compléxité, ainsi que des mesures de performance.

# 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).

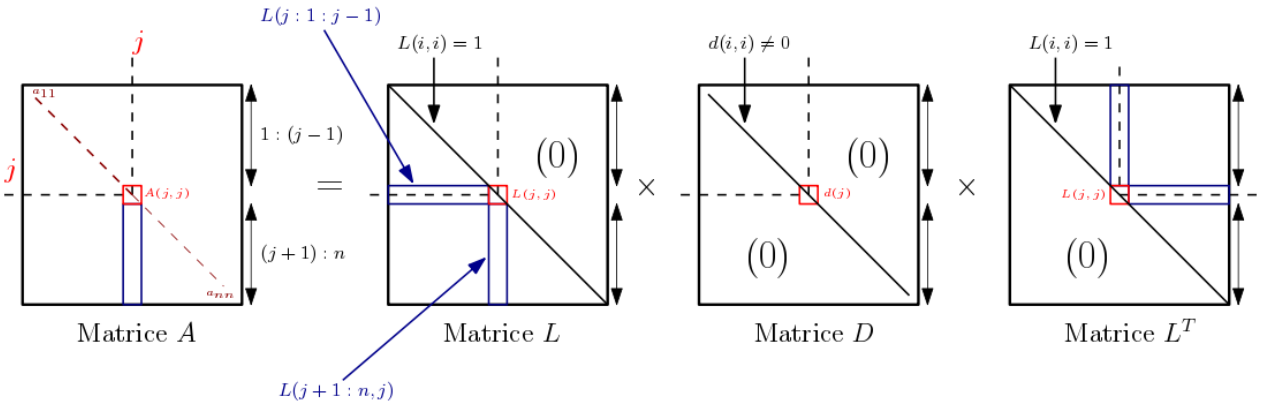


Figure 1.1: Description de l'algorithme de factorisation $A = LDL^T$.

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 $D$. Cela permet de réduire le stockage mémoire.

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 d'avantage le stockage mémoire lors de la factorisation $LDL^T$ de la matrice $A$, comme on écrase les cases mémoire au lieu de les ajouter.

En effet, on a $\frac{n^2}{2}$ **occupaction mémoire**, car on écrase la matrice $A$ par $L$ et la diagonale de $A$ par $d$.

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)-L(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 $LDL^T$:

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

Au total, on a $(\frac{n^3}{3}+n^2-\frac{4n}{3})$ opérations. Donc la complexité de l'algorithme évolue en $O(\frac{n^3}{3})$, 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 toujours vérifier que A soit symétrique ($A = A^T$) en calculant tout d'abord la matrice $w = rand(3,3)$, puis en posant $A = w \times w^T$.

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

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).

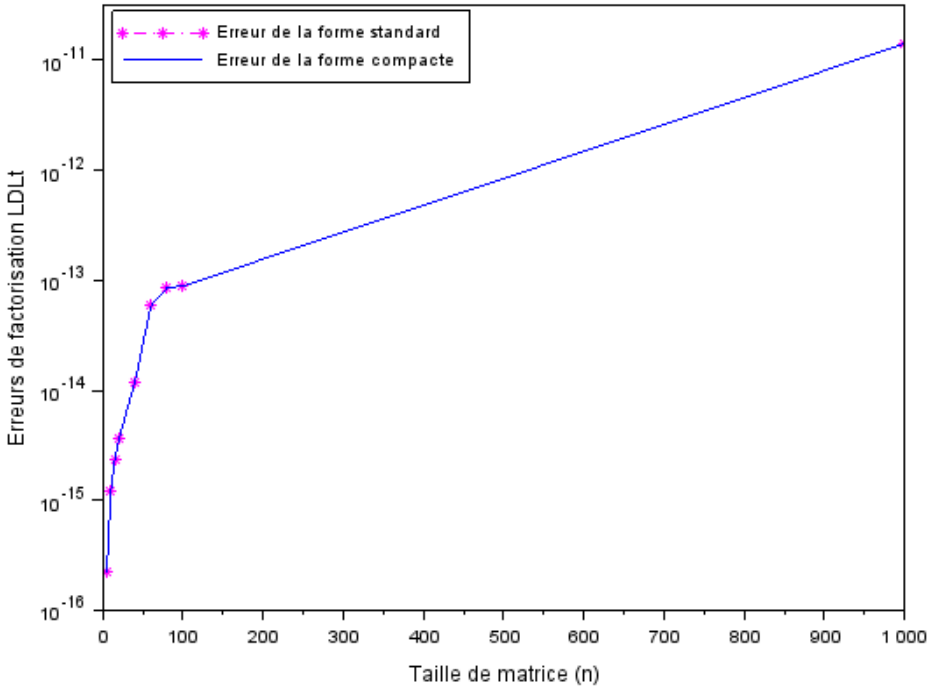


Figure 1.2: Erreurs relatives de factorisation $A = LDL^T$.

Dans un deuxième temps, on calcule la moyenne du temps de calcul des algorithmes sous formes standard et compacte, comme montré dans le tableau (1.2).

| **Taille $n$** | **Temps de la forme standard (s)** | **Temps de la forme compacte (s)** |
|---|---|---|
| 3 | 0.000115 | 0.000081 |
| 5 | 0.000154 | 0.0001050 |
| 10 | 0.000264 | 0.00023 |
| 15 | 0.000393 | 0.000307 |
| 20 | 0.000515 | 0.000462 |
| 40 | 0.001736 | 0.001527 |
| 60 | 0.0030800 | 0.0027040 |
| 80 | 0.006853 | 0.0049450 |
| 100 | 0.0115390 | 0.0068600 |
| 1000 | 5.090209 | 1.3096240 |

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

Le temps de calcul est moins dans le cas d'algorithme de factorisation $LDL^T$ sous sa forme compacte. La différence entre les temps de calcul des formes standard et compacte devient importante pour de grandes tailles de matrice $n$.

Par exemple, pour $n = 1000$ et en utilisant la forme compacte au lieu de celle standard, on diminue le temps de calcul d'un facteur approximatif de 4.

La figure (1.3) illustre la complexité temporelle des algorithmes de factorisation $A = LDL^T$.

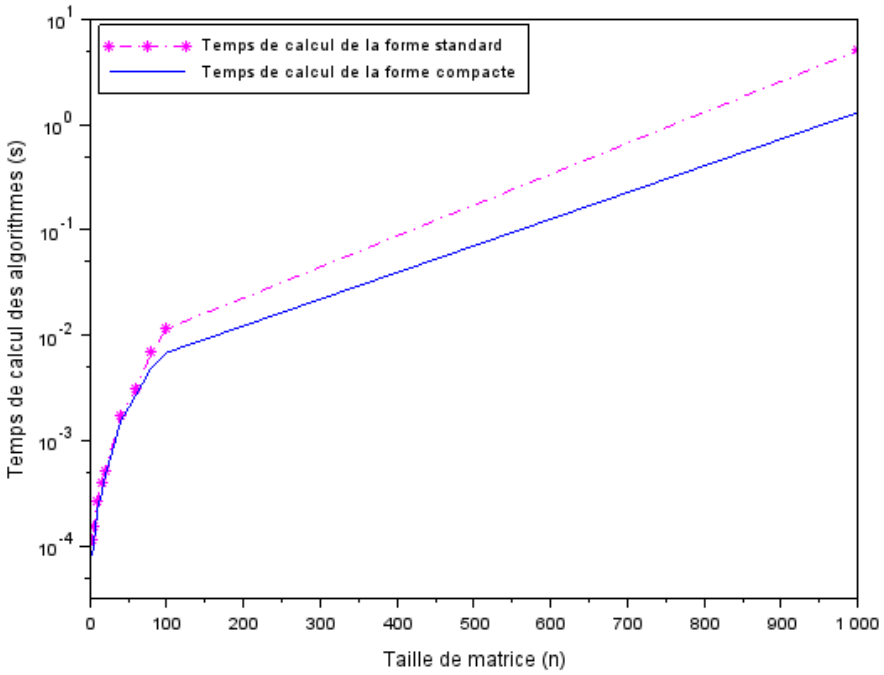


Figure 1.3: Complexité temporelle des algorithmes de factorisation $A = LDL^T$.

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 (1.3).

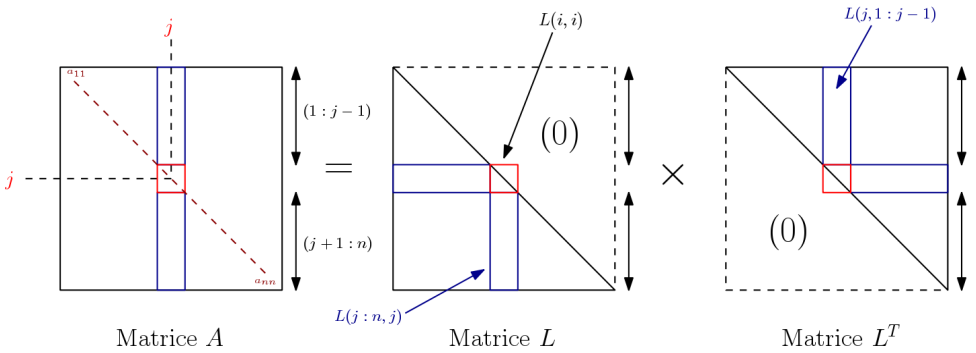


Figure 1.4: Description de l'algorithme de factorisation $A = LL^T$.

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, ainsi que le temps de calcul car on calcule uniquement la matrice $L$.
Par exemple, dans le cas de la matrice $A$ étudiée, une comparison entre la factorisation $LDL^T$ de la matrice $A$ sous ses formes standard et compacte, ainsi que la factorisation $LL^T$ de Cholesky induit, d'après le tableau (1.3), une réduction de:

- temps de calcul (Cholesky) = $\frac{5}{8}\times$ temps de calcul ($LDL^T$, forme compacte),
- temps de calcul (Cholesky) = $\frac{1}{4}\times$ temps de calcul ($LDL^T$, forme standard).

| $LDL^T$ **standard** | $LDL^T$ **compacte** | **Cholesky** $LL^T$ |
|---|---|---|
| 0.000333 | 0.000132 | 0.00008 |

Table 1.3: Comparison de la complexité temporelle entre la factorisation $LDL^T$ et Cholesky $LL^T$ de la matrice $A$ en secondes.

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

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

Au total, on a $(\frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6})$ 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 $LU$.

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 1.4: Comparison de la complexité asymptotique O entre les factorisation $LU$, $LDL^T$ et $LL^T$.

D'après le tableau (1.4), 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 standard** | $LDL^T$**, forme compacte** |
|---|---|---|---|
| 5 | 0.000348 | 0.000171 | 0.000211 |
| 10 | 0.001003 | 0.0004360 | 0.00032 |
| 20 | 0.004469 | 0.0007010 | 0.000632 |
| 40 | 0.036776 | 0.002176 | 0.002045 |
| 60 | 0.1153720 | 0.004131 | 0.003558 |
| 80 | 0.279546 | 0.007397 | 0.005799 |
| 100 | 0.50551 | 0.01151 | 0.0083650 |
| 1000 | 550.89641 | 5.398309 | 1.5236880 |

Table 1.5: 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 (1.4).

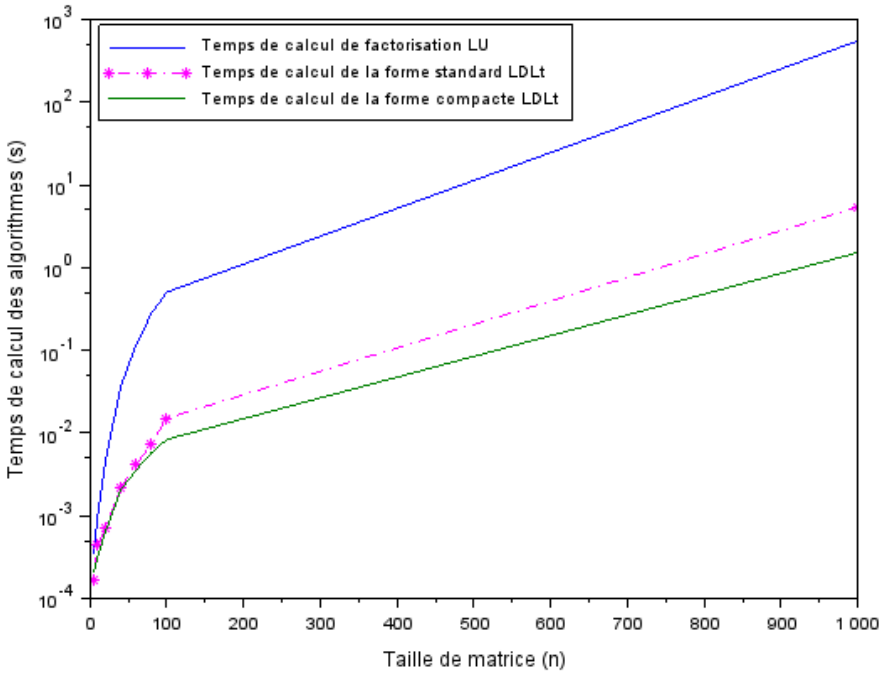


Figure 1.5: Comparison de la complexité temporelle des algorithmes de factorisation d'une matrice $A$ symétrique.

Le tableau (1.6) montre la variation des erreurs 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 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 | 6.194D+10 |
| 1000 | | | |

Table 1.6: 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 (1.5). 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 résultats d'arrondi.

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.

- 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 $O(\frac{2n^3}{3})$,
- 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 recourt à des matrices symétriques définies positives, *"Chapitre 3 Systèmes d'équations"* 2015.

## 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 format 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 $JR \in R^{1 \times nnz}$ et $JC \in R^{1 \times nnz}$, 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 1.7: 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 1.8: 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. 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 en listing 1.2.

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 1.3.

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 1.1

[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 Scilab, on obtient la matrice transposée de $A$ suivante (de 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 1.9: 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 1.5 à 1.7. Ils se basent sur $nnz$ le nombre des éléments non nuls d'une matrice creuse $A$ (listing 1.4).

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 listing1.8.

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
```

1/ **Produit Matrice creuse $A$ (Format de stockage COO) et vecteur $x$:** Soit $A \in R^{m \times n}$ une matrice creuse et $x \in R^{n \times 1}$. On cherche à calculer le produit $y = A \times x$. Donc, le vecteur résultat $y \in R^{m \times 1}$.

L'algorithme dans listing 1.9 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 1.4 et de stockage COO de la matrice creuse $A$ dans listing 1.5.

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 1.10.

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 1.9 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 1.4.

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

Table 1.10: Moyenne de temps de calcul de l'algorithme Produit Matrice creuse Vecteur.

2/ **Produit Matrice creuse $A$ (Format de stockage CSR) et vecteur $x$:** L'algorithme dans listing 1.11 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 1.4 et de stockage CSR de la matrice creuse $A$ dans listing 1.6.

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)
```

L'algorithme dans listing 1.11 est validé et le temps de calcul du produit matrice creuse vecteur est comparé dans le tableau 1.5. La comparison entre les deux formats de stockage COO et CSR est donnée dans le tableau 1.6, *"Matrice Creuse"* 2020.

| **Stockage** | **Le stockage sous forme COO** | **Le stockage sous forme CSR** |
|---|---|---|
| **Avantages** | - Facilité d'implémentation,<br>-Stockage pratique (sous forme de triplet (i,j,élément)) | - Format compressé (IA est de taille $(n+1)$ au lieu de $nnz$),<br>- 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

# References

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

*"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.

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

# Appendix A

# Appendix Chapter

Le TP4 du calcul numérique est déposé 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**