Application en Ingénierie et Programmation Numérique

"Rendu I - Méthodes Directes"

VILLEDIEU Maxance et BESQUEUT Corentin

1 octobre 2023

Table des matières

Chapitre I

Résolution de systèmes linéaires par des Méthodes Directes : Méthode de Gauss

Dans le cadre de ce premier TP, nous devions implémenter l'algorithme du *Pivot de Gauss* en utilisant le langage de programmation C.
Afin de rendre ce document plus compréhensible et lisible, nous estimons que la présence de nos codes sources en clair est nécessaire.

I.1 Détail de l'algorithme

Soient deux matrices $A \in \mathcal{M}_{m,m}$ et $b \in \mathcal{M}_{m,1}$.
L'algorithme de Gauss se décrit ainsi :

Pour $k = 1, \ldots, n-1$ Faire :

Pour $i = k+1, \ldots, n$ Faire :

$$\alpha_i^{(k)} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$$

Pour $j = k, \ldots, n$ Faire :

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \alpha_i^{(k)} a_{kj}^{(k)}$$

FIN Pour j

$$b_i^{(k+1)} = b_i^{(k)} - \alpha_i^{(k)} b_k^{(k)}$$

FIN Pour i

FIN Pour k

Une fois la matrice échelonnée par cet algorithme, on appliquera la formule suivante pour trouver les solutions du système :

$$x_n = \frac{b_n}{a_{m,m}}$$

et

$$\forall i = n-1, \dots, 1,\ x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1+i}^{n} a_{ij} x_j \right)$$

La complexité temporelle de cet algorithme est cubique soit $O(n^3)$ avec une complexité exacte de $\frac{2n^3}{3}$. Pour l'implémentation de cet algorithme, nous allons présenter deux façons de le conceptualiser avec une comparaison algorithmique des deux programmes.

I.2 Le pivot de Gauss en pratique

I.2.1 De manière générale

Soit $A \in \mathcal{M}_{m,m}$ et $B \in \mathcal{M}_{m,1}$ et x la matrice des inconnues.
Considérons alors le système suivant $Ax = b$.
Ce système peut être représenté sous la forme d'une matrice augmentée M tel que :

$$M = \left(\begin{array}{ccc|c} a_{11} & \dots & a_{1m} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{m1} & \dots & a_{mm} & b_m \end{array} \right)$$

Après exécution du pivot de Gauss, M devient

$$M = \left(\begin{array}{ccccc|c} 1 & a_{12} & a_{13} & \dots & a_{1m} & b_1^{'} \\ 0 & 1 & a_{23}^{'} & \dots & a_{2m}^{'} & b_2^{'} \\ 0 & 0 & 1 & \dots & \vdots & b_3^{'} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & 0 & \dots & 1 & b_m^{'} \end{array} \right)$$

Une fois que tous les pivots sont placés, il suffira de reconstituer le système et de le remonter afin de déterminer les inconnues comme suit :

$$\text{Soit } A^{'} = \begin{pmatrix} 1 & a_{12}^{'} & a_{13}^{'} & \dots & a_{1m}^{'} \\ 0 & 1 & a_{23}^{'} & \dots & a_{2m}^{'} \\ 0 & 0 & 1 & \dots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 1 \end{pmatrix} \text{ et } b = \begin{pmatrix} b_1^{'} \\ b_2^{'} \\ b_3^{'} \\ \vdots \\ b_m^{'} \end{pmatrix}$$

alors pour $A^{'}x = b$ on a donc $x_n = \frac{b_n}{a_{m,m}}$ et $x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1+i}^{n} a_{ij} x_j \right), \forall i = n-1, \dots, 1.$

Nous remarquerons que l'implémentation du pivot de Gauss ne nécessitera pas de mettre nos pivots à 1

I.2.2 Exercice

Résoudre le système linéaire suivant : x + y + 2z = 3
x + 2y + z = 1
2x + y + z = 0 On pose $A = \begin{pmatrix} 1 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 1 & 1 \end{pmatrix}$, $B = \begin{pmatrix} 3 \\ 1 \\ 0 \end{pmatrix}$ et $X = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$

puis la matrice augmentée $\begin{pmatrix} A & \vert & B \end{pmatrix} = \left(\begin{array}{ccc|c} 1 & 1 & 2 & 3 \\ 1 & 2 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{array}\right)$

Commençons par échelonner la matrice augmentée à l'aide du pivot de Gauss :

$$\left(\begin{array}{ccc|c} 1 & 1 & 2 & 3 \\ 1 & 2 & 1 & 1 \\ 2 & 1 & 1 & 0 \end{array}\right) \xrightarrow[L_2-L_1\to L_2]{-(L_3+2L_1)\to L_3} \left(\begin{array}{ccc|c} 1 & 1 & 2 & 3 \\ 0 & 1 & -1 & -2 \\ 0 & 1 & 3 & 6 \end{array}\right) \xrightarrow{\frac{L_3-L_2}{4}\to L_3} \left(\begin{array}{ccc|c} 1 & 1 & 2 & 3 \\ 0 & 1 & -1 & -2 \\ 0 & 0 & 1 & 2 \end{array}\right)$$

Maintenant que notre matrice augmentée est échelonnée, nous pouvons déterminer les inconnues du système par substitution (en partant du bas) :

$$\left(\begin{array}{ccc|c} 1 & 1 & 2 & 3 \\ 0 & 1 & -1 & -2 \\ 0 & 0 & 1 & 2 \end{array}\right) \xrightarrow[L_1-2L_3\to L_1]{L_2+L_3\to L_2} \left(\begin{array}{ccc|c} 1 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \end{array}\right) \xrightarrow{L_1-L_2\to L_1} \left(\begin{array}{ccc|c} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \end{array}\right)$$

Nous avons maintenant $A = I_3$ et donc nous pouvons remplacer dans le système $AX = B$, A par I_3, ce qui nous donne :

$$AX = B \iff I_3X = B \iff \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \\ 2 \end{pmatrix} \iff \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \\ 2 \end{pmatrix}$$

Nous avons alors notre couple solution du système, qui est :

x = -1
y = 0
z = 2

I.3 Implémentation de l'algorithme de Gauss en passant par le système d'équations linéaires

Dans cette partie, vous trouverez quelques commentaires sur mon implémentation de l'élimination de Gauss, ainsi que le code de l'algorithme décrit en I.1. Il est à noter que cette première implémentation ne recourt pas à l'utilisation de la matrice augmentée. En effet, le programme fonctionne directement avec le système d'équations linéaires $Ax = B$. Vous trouverez également le code de la fonction qui, a un système matriciel échelonné, retourne un vecteur solution du système.

I.3.1 Commentaires fonctionnels

Fonctions usuelles de manipulation de matrices

Ce code implémente diverses fonctions pour travailler avec des matrices à coefficients en nombre flottants.

- La fonction ***createMatrix*** alloue dynamiquement de la mémoire pour créer une matrice de nombres flottants avec un nombre spécifié de lignes et de colonnes.
- La fonction ***printMatrix*** affiche les éléments d'une matrice de nombres flottants.
- La fonction ***freeMatrix*** libère la mémoire allouée pour une matrice de nombres flottants.
- La fonction ***completeMatrix*** permet à l'utilisateur de saisir des valeurs pour remplir les éléments d'une matrice de nombres flottants.
- La fonction ***generateB*** génère un vecteur colonne B en fonction de la somme des éléments de chaque ligne de la matrice A.

Fonctions résolvant notre système linéaire $Ax = B$ à l'aide de l'algorithme de Gauss

Dans le cadre de notre résolution de systèmes d'équations linéaires, deux fonctions jouent un rôle clef dans ce code : la fonction ***gauss*** et la fonction ***resolution***.

- La fonction ***gauss*** joue un rôle important dans la préparation de la résolution de notre système d'équations linéaires. En effectuant l'élimination de Gauss sur la matrice A, elle la transforme en une matrice triangulaire supérieure. Cela signifie que les éléments sous la diagonale principale de la matrice deviennent tous des zéros, simplifiant ainsi la résolution du système. De plus, la fonction met également à jour la matrice B en conséquence, garantissant que notre système $Ax = B$ reste équilibré.

- La fonction ***resolution***, quant à elle, prend en charge la résolution effective du système linéaire une fois que la matrice A a été triangulée par la fonction **gauss**. Elle utilise la méthode de substitution pour calculer la solution et stocke le résultat dans le vecteur X. Cette étape finale permet d'obtenir les valeurs des variables inconnues du système, fournissant ainsi la solution recherchée pour le problème initial.

En combinant ces deux fonctions avec celles sus-citées en I.3.1, le code réalise un processus complet de résolution de systèmes d'équations linéaires de manière efficace et précise (aux erreurs d'arrondies près).

I.3.2 Code source

Puisque nous sommes contraints de minimiser la présence de code dans ce rapport, nous ne présenterons pas ici les fonctions usuelles de manipulations de matrices suivantes : ***createMatrix***, ***printMatrix***, ***freeMatrix***, ***completeMatrix***, ***generateB***.

```
/*
*PERFORM GAUSSIAN ELIMINATION ON A Ax=B MATRIX SYSTEM OF LINEAR EQUATIONS
*/

void gauss(float** matA, float** matb, int size){
        for(int k=0; k<size-1; k++){
                for(int i=k+1; i<size; i++){
                        float alpha=matA[i][k]/matA[k][k];
                        for(int j=k; j<size; j++){
                                matA[i][j]=matA[i][j]-alpha*matA[k][j];
                        }
                        matb[i][0]=matb[i][0]-alpha*matb[k][0];
                }
        }
}

/*
*SOLVE A MATRIX SYSTEM OF LINEAR EQUATIONS USING BACKWARD SUBSTITUTION
*/
void resolution(float** matA, float** matb, float** matx, int size){
        matx[size-1][0]=matb[size-1][0]/matA[size-1][size-1];
        for (int i=size-2; i>=0; i--){
                float sum=0;
                for(int j=i+1; j<size; j++){
                        sum+=matA[i][j]*matx[j][0];
                }
                matx[i][0]=(1/matA[i][i])*(matb[i][0]-sum);
        }
}
```

I.3.3 Interactions Utilisateur/Console

Entrées utilisateur

En premier lieu dans notre programme, nous avons besoin de spécifier le système $Ax = B$ à l'ordinateur. Pour ce faire, nous allons dans l'ordre :

1. Allouer une matrice A en mémoire. Cette matrice verra sa taille définie par la première entrée utilisateur du programme (nous demanderons consécutivement le nombre de lignes, puis le nombre de colonnes de la matrice).
2. Définir les coefficients de la matrice A. Il s'agira de la deuxième entrée utilisateur de notre programme.
   Par définition de notre fonction ***completeMatrix***, nous remplirons la matrice dans l'ordre suivant :
   $a_{1,1}, a_{1,2}, ..., a_{1,n},$ puis $a_{2,1}, ...a_{2,n},$ jusque $a_{n,1}, ..., a_{n,n}$
3. Allouer une matrice B en mémoire. À noter que la taille de B est définie automatiquement en fonction de la taille de A. Nous avons $A \in \mathcal{M}_{n,p} \Rightarrow B \in \mathcal{M}_{n,1}$.
4. Définir les coefficients de la matrice B. Chaque coefficient prendra la valeur de la somme des éléments de la ligne respective de la matrice A.
   Nous avons donc :
   Soient $A \in \mathcal{M}_{n,p}$ et $B \in \mathcal{M}_{n,1}, \forall i \in \{1, n\}, b_{i,1} = \sum_{j=1}^{p} a_{i,p}$.
5. Allouer une matrice X en mémoire. Cette matrice aura la même taille que la matrice B. Ces coefficients ne seront pas définis pour le moment.

En guise d'exemple, le système matriciel $AX = B$ suivant :

$$\begin{pmatrix} 3 & 0 & 4 \\ 7 & 4 & 2 \\ -1 & 1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 13 \\ 2 \end{pmatrix} \tag{I.1}$$

est représenté par l'entrée utilisateur :

Listing I.1 – User Input

```
Row count of matrix A : 3

Column count of matrix A : 3

                    FILL IN THE VALUE OF MATRIX A

Value for a_1,1:   3
Value for a_1,2:   0
Value for a_1,3:   4
Value for a_2,1:   7
Value for a_2,2:   4
Value for a_2,3:   2
Value for a_3,1:   -1
Value for a_3,2:   1
Value for a_3,3:   2
```

Une fois toutes les matrices initialisées et complétées, nous pouvons attaquer la résolution du système par la triangularisation du système. Ceci fait, nous résolverons le système obtenu pour obtenir notre vecteur X solution.

Affichage Console

Dès lors le système $AX = B$ connu par l'ordinateur, ce dernier peut retrouver les valeurs de la matrice X. Voici l'affichage produit par notre programme en console :

Listing I.2 – Console Display of the Gauss elimination for the AX

```
                    A matrix

3.000000    0.000000    4.000000
7.000000    4.000000    2.000000
-1.000000    1.000000    2.000000

                    B matrix

7.000000
13.000000
2.000000

                    TRIANGULARIZATION
                    A Matrix

3.000000    0.000000    4.000000
0.000000    4.000000    -7.333333
0.000000    0.000000    5.166667

                    B Matrix

7.000000
-3.333332
```

```
5.166667

                    SOLVING
                    SOLUTION VECTOR X

1.000000
1.000000
1.000000
```

Il est à repérer que le programme affiche dans cet ordre :

- La **Matrice** A
- La **Matrice** B
- La **Matrice** A une fois triangulée supérieure
- La **Matrice** B une fois mise à jour en conséquence pour que le système reste équilibré
- La **Matrice** X solution du système

Remarque : le temps d'exécution de ce programme a été de 0.000237 secondes

I.3.4 Exemples d'exécution

Note : Par souci de présentation, les coefficients des matrices sont ici arrondis pour une précision de 10^{-3}

Soient les matrices A données dans le TP et en annexe du document. On obtient respectivement les résultats suivants :

Listing I.3 – $A_1X = B$ results

```
                                        A matrix
        3.000    0.000    4.000
        7.000    4.000    2.000
        -1.000    1.000    2.000

                                        B matrix
        7.000
        13.000
        2.000
                                        TRIANGULARIZATION

                                        A Matrix
        3.000    0.000    4.000
        0.000    4.000    -7.333
        0.000    0.000    5.167

                                        B Matrix
        7.000
        -3.333
        5.167
                                        SOLVING

                                        SOLUTION VECTOR X
        1.000
        1.000
        1.000

        Temps d'execution : 0.000181 secondes
```

Listing I.4 – $A_2X = B$ results

```
                                        A matrix
        -3.000    3.000    -6.000
        -4.000    7.000    8.000
```

```
        5.000    7.000    -9.000

                                    B matrix
        -6.000
        11.000
        3.000
                                    TRIANGULARIZATION

                                    A Matrix
        -3.000    3.000    -6.000
        0.000    3.000    16.000
        0.000    0.000    -83.000

                                    B Matrix
        -6.000
        19.000
        -83.000
                                    SOLVING

                                    SOLUTION VECTOR X
        1.000
        1.000
        1.000

        Temps d'execution : 0.000209 secondes
```

Listing I.5 – $A_3X = B$ results

```
                                    A matrix
        4.000    1.000    1.000
        2.000    -9.000    0.000
        0.000    -8.000    6.000

                                    B matrix
        6.000
        -7.000
        -2.000
                                    TRIANGULARIZATION

                                    A Matrix
        4.000    1.000    1.000
        0.000    -9.500    -0.500
        0.000    0.000    6.421

                                    B Matrix
        6.000
        -10.000
        6.421
                                    SOLVING

                                    SOLUTION VECTOR X
        1.000
        1.000
        1.000

        Temps d'execution : 0.000195 secondes
```

Listing I.6 – $A_4X = B$ results

```
                                    A matrix
        7.000    6.000    9.000
        4.000    5.000    -4.000
        -7.000    -3.000    8.000

```

```
                              B matrix
     22.000
     5.000
     -2.000
                              TRIANGULARIZATION

                              A Matrix
     7.000    6.000    9.000
     0.000    1.571    -9.143
     0.000    0.000    34.455

                              B Matrix
     22.000
     -7.571
     34.455
                              SOLVING

                              SOLUTION VECTOR X
     1.000
     1.000
     1.000

     Temps d'execution : 0.000367 secondes
```

Listing I.7 – $A_5X = B$ results

```
                              A matrix
     1.000    0.500    0.250
     0.500    1.000    0.000
     0.250    0.000    1.000

                              B matrix
     1.750
     1.500
     1.250
                              TRIANGULARIZATION

                              A Matrix
     1.000    0.500    0.250
     0.000    0.750    -0.125
     0.000    0.000    0.917

                              B Matrix
     1.750
     0.625
     0.917
                              SOLVING

                              SOLUTION VECTOR X
     1.000
     1.000
     1.000

     Temps d'execution : 0.000206 secondes
```

Listing I.8 – $A_6X = B$ results

```
                              A matrix
     1.000    0.500    0.250    0.125    0.062    0.031
     0.500    1.000    0.000    0.000    0.000    0.000
     0.250    0.000    1.000    0.000    0.000    0.000
     0.125    0.000    0.000    1.000    0.000    0.000
     0.062    0.000    0.000    0.000    1.000    0.000
     0.031    0.000    0.000    0.000    0.000    1.000
```

```

                                B matrix
     1.969
     1.500
     1.250
     1.125
     1.062
     1.031
                                TRIANGULARIZATION

                                A Matrix
     1.000   0.500   0.250   0.125   0.062   0.031
     0.000   0.750  -0.125  -0.062  -0.031  -0.016
     0.000   0.000   0.917  -0.042  -0.021  -0.010
     0.000   0.000   0.000   0.977  -0.011  -0.006
     0.000   0.000   0.000   0.000   0.994  -0.003
     0.000   0.000   0.000   0.000   0.000   0.999

                                B Matrix
     1.969
     0.516
     0.844
     0.960
     0.991
     0.999
                                SOLVING

                                SOLUTION VECTOR X
     1.000
     1.000
     1.000
     1.000
     1.000
     1.000

     Temps d'execution : 0.000201 secondes
```

Listing I.9 – $A_7X = B$ results

```
                                A matrix
     1.000   0.500   0.250   0.125   0.062   0.031   0.016   0.008
     0.500   1.000   0.000   0.000   0.000   0.000   0.000   0.000
     0.250   0.000   1.000   0.000   0.000   0.000   0.000   0.000
     0.125   0.000   0.000   1.000   0.000   0.000   0.000   0.000
     0.062   0.000   0.000   0.000   1.000   0.000   0.000   0.000
     0.031   0.000   0.000   0.000   0.000   1.000   0.000   0.000
     0.016   0.000   0.000   0.000   0.000   0.000   1.000   0.000
     0.008   0.000   0.000   0.000   0.000   0.000   0.000   1.000

                                B matrix
     1.992
     1.500
     1.250
     1.125
     1.062
     1.031
     1.016
     1.008
                                TRIANGULARIZATION

                                A Matrix
     1.000   0.500   0.250   0.125   0.062   0.031   0.016   0.008
     0.000   0.750  -0.125  -0.062  -0.031  -0.016  -0.008  -0.004
     0.000   0.000   0.917  -0.042  -0.021  -0.010  -0.005  -0.003
     0.000   0.000   0.000   0.977  -0.011  -0.006  -0.003  -0.001
     0.000   0.000   0.000   0.000   0.994  -0.003  -0.001  -0.001
     0.000   0.000   0.000   0.000   0.000   0.999  -0.001  -0.000
     0.000   0.000   0.000   0.000   0.000   0.000   1.000  -0.000
     0.000   0.000   0.000   0.000   0.000   0.000   0.000   1.000

                                B Matrix
```

```
          1.992
          0.504
          0.836
          0.956
          0.989
          0.997
          0.999
          1.000
                              SOLVING

                              SOLUTION VECTOR X
          1.000
          1.000
          1.000
          1.000
          1.000
          1.000
          1.000
          1.000

          Temps d'execution : 0.000329 secondes
```

Listing I.10 – $A_8X = B$ results

```
                              A matrix
          1.000   0.500   0.250   0.125   0.062   0.031   0.016   0.008   0.004   0.002
          0.500   1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
          0.250   0.000   1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
          0.125   0.000   0.000   1.000   0.000   0.000   0.000   0.000   0.000   0.000
          0.062   0.000   0.000   0.000   1.000   0.000   0.000   0.000   0.000   0.000
          0.031   0.000   0.000   0.000   0.000   1.000   0.000   0.000   0.000   0.000
          0.016   0.000   0.000   0.000   0.000   0.000   1.000   0.000   0.000   0.000
          0.008   0.000   0.000   0.000   0.000   0.000   0.000   1.000   0.000   0.000
          0.004   0.000   0.000   0.000   0.000   0.000   0.000   0.000   1.000   0.000
          0.002   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   1.000

                              B matrix
          1.998
          1.500
          1.250
          1.125
          1.062
          1.031
          1.016
          1.008
          1.004
          1.002
                              TRIANGULARIZATION

                              A Matrix
          1.000   0.500   0.250   0.125   0.062   0.031   0.016   0.008   0.004   0.002
          0.000   0.750   -0.125   -0.062   -0.031   -0.016   -0.008   -0.004   -0.002   -0.001
          0.000   0.000   0.917   -0.042   -0.021   -0.010   -0.005   -0.003   -0.001   -0.001
          0.000   0.000   0.000   0.977   -0.011   -0.006   -0.003   -0.001   -0.001   -0.000
          0.000   0.000   0.000   0.000   0.994   -0.003   -0.001   -0.001   -0.000   -0.000
          0.000   0.000   0.000   0.000   0.000   0.999   -0.001   -0.000   -0.000   -0.000
          0.000   0.000   0.000   0.000   0.000   0.000   1.000   -0.000   -0.000   -0.000
          0.000   0.000   0.000   0.000   0.000   0.000   0.000   1.000   -0.000   -0.000
          0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   1.000   -0.000
          0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   1.000

                              B Matrix
          1.998
          0.501
          0.834
          0.955
          0.989
          0.997
          0.999
          1.000
          1.000
          1.000
                              SOLVING

                              SOLUTION VECTOR X
          1.000
          1.000
          1.000
          1.000
          1.000
```

```
1.000
1.000
1.000
1.000
1.000

Temps d'execution : 0.000360 secondes
```

Listing I.11 – $A_9X = B$ results

```
                                        A matrix
3.000     -1.000     0.000
-2.000     3.000     -1.000
0.000     -2.000     3.000

                                        B matrix
2.000
0.000
1.000
                                        TRIANGULARIZATION

                                        A Matrix
3.000     -1.000     0.000
0.000     2.333     -1.000
0.000     0.000     2.143

                                        B Matrix
2.000
1.333
2.143
                                        SOLVING

                                        SOLUTION VECTOR X
1.000
1.000
1.000

Temps d'execution : 0.000345 secondes
```

Listing I.12 – $A_{10} = B$ results

```
                                        A matrix
3.000     -1.000     0.000     0.000     0.000     0.000
-2.000     3.000     -1.000     0.000     0.000     0.000
0.000     -2.000     3.000     -1.000     0.000     0.000
0.000     0.000     -2.000     3.000     -1.000     0.000
0.000     0.000     0.000     -2.000     3.000     -1.000
0.000     0.000     0.000     0.000     -2.000     3.000

                                        B matrix
2.000
0.000
0.000
0.000
0.000
1.000
                                        TRIANGULARIZATION

                                        A Matrix
3.000     -1.000     0.000     0.000     0.000     0.000
0.000     2.333     -1.000     0.000     0.000     0.000
0.000     0.000     2.143     -1.000     0.000     0.000
0.000     0.000     0.000     2.067     -1.000     0.000
0.000     0.000     0.000     0.000     2.032     -1.000
0.000     0.000     0.000     0.000     0.000     2.016
```

```
                                    B Matrix
     2.000
     1.333
     1.143
     1.067
     1.032
     2.016
                                    SOLVING

                                    SOLUTION VECTOR X
     1.000
     1.000
     1.000
     1.000
     1.000
     1.000

     Temps d'execution : 0.000407 secondes
```

Listing I.13 – $A_{11}X = B$ results

```
                                    A matrix
     3.000   -1.000   0.000   0.000   0.000   0.000   0.000   0.000
     -2.000   3.000   -1.000   0.000   0.000   0.000   0.000   0.000
     0.000   -2.000   3.000   -1.000   0.000   0.000   0.000   0.000
     0.000   0.000   -2.000   3.000   -1.000   0.000   0.000   0.000
     0.000   0.000   0.000   -2.000   3.000   -1.000   0.000   0.000
     0.000   0.000   0.000   0.000   -2.000   3.000   -1.000   0.000
     0.000   0.000   0.000   0.000   0.000   -2.000   3.000   -1.000
     0.000   0.000   0.000   0.000   0.000   0.000   -2.000   3.000

                                    B matrix
     2.000
     0.000
     0.000
     0.000
     0.000
     0.000
     0.000
     1.000
                                    TRIANGULARIZATION

                                    A Matrix
     3.000   -1.000   0.000   0.000   0.000   0.000   0.000   0.000
     0.000   2.333   -1.000   0.000   0.000   0.000   0.000   0.000
     0.000   0.000   2.143   -1.000   0.000   0.000   0.000   0.000
     0.000   0.000   0.000   2.067   -1.000   0.000   0.000   0.000
     0.000   0.000   0.000   0.000   2.032   -1.000   0.000   0.000
     0.000   0.000   0.000   0.000   0.000   2.016   -1.000   0.000
     0.000   0.000   0.000   0.000   0.000   0.000   2.008   -1.000
     0.000   0.000   0.000   0.000   0.000   0.000   0.000   2.004

                                    B Matrix
     2.000
     1.333
     1.143
     1.067
     1.032
     1.016
     1.008
     2.004
                                    SOLVING

                                    SOLUTION VECTOR X
     1.000
     1.000
     1.000
     1.000
     1.000
     1.000
     1.000
     1.000

     Temps d'execution : 0.000318 secondes
```

Listing I.14 – $A_{12} = B$ results

```
                                        A matrix
    3.000   -1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
   -2.000    3.000  -1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
    0.000   -2.000   3.000  -1.000   0.000   0.000   0.000   0.000   0.000   0.000
    0.000    0.000  -2.000   3.000  -1.000   0.000   0.000   0.000   0.000   0.000
    0.000    0.000   0.000  -2.000   3.000  -1.000   0.000   0.000   0.000   0.000
    0.000    0.000   0.000   0.000  -2.000   3.000  -1.000   0.000   0.000   0.000
    0.000    0.000   0.000   0.000   0.000  -2.000   3.000  -1.000   0.000   0.000
    0.000    0.000   0.000   0.000   0.000   0.000  -2.000   3.000  -1.000   0.000
    0.000    0.000   0.000   0.000   0.000   0.000   0.000  -2.000   3.000  -1.000
    0.000    0.000   0.000   0.000   0.000   0.000   0.000   0.000  -2.000   3.000

                                        B matrix
    2.000
    0.000
    0.000
    0.000
    0.000
    0.000
    0.000
    0.000
    0.000
    1.000
                                        TRIANGULARIZATION

                                        A Matrix
    3.000   -1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
    0.000    2.333  -1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
    0.000    0.000   2.143  -1.000   0.000   0.000   0.000   0.000   0.000   0.000
    0.000    0.000   0.000   2.067  -1.000   0.000   0.000   0.000   0.000   0.000
    0.000    0.000   0.000   0.000   2.032  -1.000   0.000   0.000   0.000   0.000
    0.000    0.000   0.000   0.000   0.000   2.016  -1.000   0.000   0.000   0.000
    0.000    0.000   0.000   0.000   0.000   0.000   2.008  -1.000   0.000   0.000
    0.000    0.000   0.000   0.000   0.000   0.000   0.000   2.004  -1.000   0.000
    0.000    0.000   0.000   0.000   0.000   0.000   0.000   0.000   2.002  -1.000
    0.000    0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   2.001

                                        B Matrix
    2.000
    1.333
    1.143
    1.067
    1.032
    1.016
    1.008
    1.004
    1.002
    2.001
                                        SOLVING

                                        SOLUTION VECTOR X
    1.000
    1.000
    1.000
    1.000
    1.000
    1.000
    1.000
    1.000
    1.000
    1.000

    Temps d'execution : 0.000563 secondes
```

Nous remarquerons que sur le calcul de A_4, par exemple, nous tombons sur des valeurs extrêmement proches de 1. Ceci est provoqué à cause des erreurs d'arrondis provoquées par l'encodage des nombres flottants.

I.4 Pivot de Gauss avec matrice augmentée

I.4.1 Code source

Voici le code source de mon implémentation du pivot de Gauss via le passage par la matrice augmentée.
C'est-à-dire que dans mon implémentation, on fera usage de la concaténation des matrices.

```c
/*
 * BUILD AUGMENTED MATRIX
 */
float **AugmentedMatrix(float **M1, float **M2, int m, int n) {
  float **A = allocate(m, m + 1);
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n + 1; j++) {
      (j != n) ? (A[i][j] = M1[i][j]) : (A[i][j] = M2[i][0]);
    }
  }
  return A;
}
/*
 * PERFORM GAUSS ALGORITHM ONLY ON AUGMENTED MATRIX
 */
void gauss(float **A, int m, int p) {
  if (m != p) {
    puts("La matrice doit etre carree !");
    return;
  }
  for (int k = 0; k <= m - 1; k++) {
    for (int i = k + 1; i < m; i++) {
      float pivot = A[i][k] / A[k][k];
      for (int j = k; j <= m; j++) {
        A[i][j] = A[i][j] - pivot * A[k][j];
      }
    }
  }
}
/*
 * DETERMINE ALL UNKNOWNS VARIABLES
 */
float *findSolutions(float **A, int m) {
  float *S = calloc(m, sizeof *S);
  S[m - 1] = A[m - 1][m] / A[m - 1][m - 1];
  for (int i = m - 1; i >= 0; i--) {
    S[i] = A[i][m];
    for (int j = i + 1; j < m; j++) {
      S[i] -= A[i][j] * S[j];
    }
    S[i] = S[i] / A[i][i];
  }
  return S;
}
```

I.4.2 Commentaires du code

Mon implémentation utilise strictement l'algorithme de Gauss rappelé précédemment avec seulement quelques changements d'indices puisque au lieu de travailler sur une matrice carrée et un vecteur colonne, mon programme utilise une matrice augmentée ayant m lignes et $m+1$ colonnes, $m \in \mathbb{N}^*$.
Détail des fonctions non conventionnelles :
Comme mentionné précédemment, je ne détaillerai pas les fonctions gauss() et findSolutions() puisque ces fonctions permettent seulement d'une part d'implémenter l'algorithme de Gauss et d'autre part à "remonter" la matrice échelonnée afin de récupérer les valeurs des inconnus.

-float **AugmentedMatrix(float **M1, float **M2, int m, int n) : cette fonction permet de créer une matrice $A \in \mathcal{M}_{m,m+1}$ à partir de la concaténation de $M1 \in \mathcal{M}_{mm}$ et $M2 \in \mathcal{M}_{m,1}$.
Soient a_{ij} les coefficients peuplant A, b_{ij} les coefficients peuplant $M1$ et c_{i0} les coefficients peuplant $M2$.
On obtient alors $a_{ij} = b_{ij} \forall i \in \mathbb{N}_m, \forall j \in \mathbb{N}_m$ et $a_{ij} = c_{i0}$ si $j = m + 1$.
Cette fonction renvoie alors A, la matrice de flottants créée dynamiquement.
Les fonctions *fillM()*, *printMatrix()*, *freeAll()* et *multiplication()* et *allocate()* sont quatre fonctions utilitaires qui permettent respectivement : de remplir une matrice, d'afficher convenablement une matrice, de libérer les matrices en mémoires, de définir la multiplication matricielle et enfin d'allouer de la mémoire pour déclarer les matrices (avec quelques légères sécurités permettant d'être sûr que les matrices sont bien instanciées convenablement).

I.4.3 Inputs / Outputs

Mon programme demande d'abord 4 entiers m, p, n, q qui correspondent aux dimensions de la première matrice $A \in \mathcal{M}_{mp}$ et de la seconde matrice $X \in \mathcal{M}_{nq}$. Le but étant de résoudre le système $AX = b$, nous initialiserons X à 1. Ce choix de valeur permettra de contrôler la validité du programme, ainsi à la fin de ce dernier, si $\forall x_i \neq 1, \forall i \in \mathbb{N}_n$, on pourra affirmer que le programme est faux.
Sur les $m \times p$ prochaines lignes, le programme demande les coefficients de A.
Sur les $n \times q$ prochaines lignes, le programme demandera les coefficients du vecteur colonne X, que l'utilisateur initialisera à 1.
On peut alors automatiser les entrées en utilisant des fichiers.

Ainsi les matrices : $M = \begin{pmatrix} 3 & 0 & 4 \\ 7 & 4 & 2 \\ -1 & 1 & 2 \end{pmatrix}$ et $X = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$

sont représentés par ce fichier d'entrées :

Listing I.15 – input.txt

```
3 3
3 1
3 0 4
7 4 2
-1 1 2
1 1 1
```

Pour ce qui est des résultats produits par mon programme, une fois injecté dans un fichier texte, une output "type" ressemble à ceci.

Listing I.16 – Gauss elimination with M and X matrix

```
PRINTING MATRIX FROM: 0x556d938672c0 LOCATION :
3.000000 0.000000 4.000000
7.000000 4.000000 2.000000
-1.000000 1.000000 2.000000
PRINTING MATRIX FROM: 0x556d93867340 LOCATION :
1.000000
1.000000
1.000000
PRINTING MATRIX FROM: 0x556d938673c0 LOCATION :
7.000000
13.000000
2.000000
PRINTING MATRIX FROM: 0x556d93867440 LOCATION :
3.000000 0.000000 4.000000 7.000000
0.000000 4.000000 -7.333333 -3.333332
0.000000 0.000000 5.166667 5.166667
SOLUTIONS
x0 = 1.000000
```

```
x1 = 1.000000
x2 = 1.000000
RUNTIME: 0.000002 seconds
```

On remarquera que le programme affiche dans cet ordre :
- La Matrice A
- La Matrice X
- La Matrice B trouvée avec les valeur de X
- La Matrice augmentée en triangle supérieur
- Les solutions
- Un timer permettant de contrôler le temps d'exécution approximatif de mon programme

I.4.4 Exemples d'exécutions

Soient $A_2 = \begin{pmatrix} -3 & 3 & -6 \\ -4 & 7 & 8 \\ 5 & 7 & -9 \end{pmatrix}$, $A_4 = \begin{pmatrix} 7 & 6 & 9 \\ 4 & 5 & -4 \\ -7 & -3 & 8 \end{pmatrix}$, $A_6 = \begin{pmatrix} -3 & 3 & -6 \\ -4 & 7 & 8 \\ 5 & 7 & -9 \end{pmatrix}$

On obtient respectivement ces résultats :

Listing I.17 – Matrix 2 results

```
PRINTING MATRIX FROM: 0x55f604fb32c0 LOCATION :
-3.000000 3.000000 -6.000000
-4.000000 7.000000 8.000000
5.000000 7.000000 -9.000000
PRINTING MATRIX FROM: 0x55f604fb3340 LOCATION :
1.000000
1.000000
1.000000
PRINTING MATRIX FROM: 0x55f604fb33c0 LOCATION :
-6.000000
11.000000
3.000000
PRINTING MATRIX FROM: 0x55f604fb3440 LOCATION :
-3.000000 3.000000 -6.000000 -6.000000
0.000000 3.000000 16.000000 19.000000
0.000000 0.000000 -83.000000 -83.000000
SOLUTIONS
x0 = 1.000000
x1 = 1.000000
x2 = 1.000000
RUNTIME: 0.000002 seconds
```

Listing I.18 – Matrix 4 results

```
PRINTING MATRIX FROM: 0x55f7afd662c0 LOCATION :
7.000000 6.000000 9.000000
4.000000 5.000000 -4.000000
-7.000000 -3.000000 8.000000
PRINTING MATRIX FROM: 0x55f7afd66340 LOCATION :
1.000000
1.000000
1.000000
PRINTING MATRIX FROM: 0x55f7afd663c0 LOCATION :
22.000000
5.000000
-2.000000
PRINTING MATRIX FROM: 0x55f7afd66440 LOCATION :
7.000000 6.000000 9.000000 22.000000
0.000000 1.571428 -9.142858 -7.571429
0.000000 0.000000 34.454552 34.454548
```

```
SOLUTIONS
x0 = 1.000001
x1 = 0.999999
x2 = 1.000000
RUNTIME: 0.000002 seconds
```

Listing I.19 – Matrix 6 results

```
PRINTING MATRIX FROM: 0x557fdaa552c0 LOCATION :
3.000000 −1.000000 0.000000
0.000000 3.000000 −1.000000
0.000000 −2.000000 3.000000
PRINTING MATRIX FROM: 0x557fdaa55340 LOCATION :
1.000000
1.000000
1.000000
PRINTING MATRIX FROM: 0x557fdaa553c0 LOCATION :
2.000000
2.000000
1.000000
PRINTING MATRIX FROM: 0x557fdaa55440 LOCATION :
3.000000 −1.000000 0.000000 2.000000
0.000000 3.000000 −1.000000 2.000000
0.000000 0.000000 2.333333 2.333333
SOLUTIONS
x0 = 1.000000
x1 = 1.000000
x2 = 1.000000
RUNTIME: 0.000002 seconds
```

On apercevra que sur le calcul de A_4, on obtient sur des valeurs extrêmement proches de 1. Ceci est provoqué par des erreurs d'arrondis issus par l'encodage des nombres flottants.
Nous remarquerons que l'implémentation utilisant la matrice augmentée est sensiblement meilleure en terme d'efficacité. En effet, le temps d'exécution est multiplié par 100 sur la première implémentation.

Chapitre II

Résolution de systèmes linéaires par des Méthodes Itératives

II.1 Méthode de Gauss-Seidel

II.1.1 Introduction à la méthode de Gauss-Seidel

La méthode de Gauss-Seidel est une méthode itérative pour résoudre les systèmes linéaires de la forme $Ax = b$, où A est une matrice carrée d'ordre n et x, b sont des vecteurs de $\mathbb{R}^n$. C'est une méthode qui génère une suite qui converge vers la solution de ce système lorsque celle-ci en a une et lorsque les conditions de convergence suivantes sont satisfaites (quels que soient le vecteur b et le point initial x^0) :

— Si la matrice A est symétrique définie positive,

— Si la matrice A est à diagonale strictement dominante.

II.1.2 Mise en place des matrices pour la méthode de Gauss-Seidel

Soit $Ax = b$ le système linéaire à résoudre, où $A \in \mathcal{M}_{n,n}$ et $b \in \mathcal{M}_{n,1}$. On cherche $x \in \mathcal{M}_{n,1}$ solution du système. Dans un premier temps, on va écrire A sous la forme $A = D - E - F$ où D est une matrice diagonale, E est une matrice triangulaire inférieure, et F est une matrice triangulaire supérieure.
On peut alors écrire :

$$Ax = b \tag{II.1}$$
$$\Leftrightarrow (D - E - F)x = b \tag{II.2}$$
$$\Leftrightarrow Dx = b - (E + F)x \tag{II.3}$$
$$\Leftrightarrow x = D^{-1}[b - (E + F)x] \tag{II.4}$$

On définit ensuite une suite de vecteurs (x^k) en choisissant un vecteur x^0 et par la formule de récurrence :

$$x_i^{k+1} = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{k+1} - \sum_{j=i+1}^{n} a_{i,j} x_j^k \right) \tag{II.5}$$

II.1.3 Algorithme

Pour résoudre un système $Ax = b$, avec $A \in \mathcal{M}_n$ et $b \in \mathcal{M}_{n,1}$, on s'appuie sur l'algorithme suivant en posant :

— un vecteur initial $x^{(0)}$ choisi au préalable,

— l'erreur à l'itération k=0 calculée par $\varepsilon^{(0)} = \|Ax^{(0)} - b\|$,
— une variable k qui sera notre compteur d'itération.

$x^{(0)} = x_0 \in \mathcal{M}_{n,1}$
$\varepsilon^{(0)} = \varepsilon$ (erreur)
$k = 0$
Tant Que $(\varepsilon^{(k)} >= \varepsilon)$ faire :
    Pour $i = 1$ a n:
        $x_i^{(k+1)} = \frac{1}{a_{i,i}}\left[b_i - \left(\sum\limits_{j=i+1}^{n} a_{i,j}x_j^{(k)} + \sum\limits_{j=1}^{i-1} a_{i,j}x_j^{(k+1)}\right)\right]$ pour $i = 1, ..., n$
    $\varepsilon^{(k+1)} = \|Ax^{(k+1)} - b\|$
    $k = k + 1$
Fin Tant Que

II.1.4 Résolution manuelle

Soit $A = \begin{pmatrix} 1 & 2 & 2 \\ 1 & 3 & 3 \\ 3 & 7 & 8 \end{pmatrix} \in \mathcal{M}_3(\mathbb{R})$, et $b = \begin{pmatrix} 2 \\ 2 \\ 8 \end{pmatrix} \in \mathcal{M}_{3,1}(\mathbb{R})$

Calculons le vecteur $x^{(1)}$ (vecteur x trouvé après 1 itération de l'algorithme) solution du système $Ax = b$, en prenant comme point initial $x^{(0)} = (0, 0, 0)$:

Résolution par le calcul itératif

Dans cette sous-partie, nous résolverons le système de la même manière que le fait l'algorithme sus-cité.
Pour obtenir le vecteur $x^{(1)}$ (obtenu à l'itération $k = 1$), il nous faut obtenir $x_1^{(1)}$, $x_2^{(1)}$,$x_3^{(1)}$ par la formule suivante :

$$x_i^{(k+1)} = \frac{1}{a_{i,i}}\left[b_i - \left(\sum_{j=i+1}^{n} a_{i,j}x_j^{(k)} + \sum_{j=1}^{i-1} a_{i,j}x_j^{(k+1)}\right)\right] \text{ pour } i = 1, ..., 3$$

Pour $i = 1$:

$$x_1^{(1)} = \frac{1}{a_{1,1}}\left[b_1 - \left(\sum_{j=2}^{3} a_{1,j}x_j^{(0)} + \sum_{j=1}^{0} a_{1,j}x_j^{(1)}\right)\right] \tag{II.6}$$

$$= \frac{1}{1}\left[2 - \left(a_{1,2}x_2^{(0)} + a_{2,2}x_2^{(0)} + 0\right)\right] \tag{II.7}$$

$$= 2 - 2 \times 0 - 3 \times 0 = 2 \tag{II.8}$$

Pour $i = 2$:

$$x_2^{(1)} = \frac{1}{a_{2,2}}\left[b_2 - \left(\sum_{j=3}^{3} a_{2,j}x_j^{(0)} + \sum_{j=1}^{1} a_{2,j}x_j^{(1)}\right)\right] \tag{II.9}$$

$$= \frac{1}{3}\left[2 - \left(a_{2,3}x_3^{(0)} + a_{2,1}x_1^{(1)}\right)\right] \tag{II.10}$$

$$= \frac{1}{3}\left(2 - 3 \times 0 - 1 \times 2\right) \tag{II.11}$$

$$= \frac{1}{3} \times 0 = 0 \tag{II.12}$$

Pour $i = 3$:

$$x_3^{(1)} = \frac{1}{a_{3,3}}\left[b_3 - \left(\sum_{j=4}^{3} a_{3,j}x_j^{(0)} + \sum_{j=1}^{2} a_{3,j}x_j^{(1)}\right)\right] \tag{II.13}$$

$$= \frac{1}{8}\left[8 - \left(0 + a_{3,1}x_1^{(1)} + a_{3,2}x_2^{(1)}\right)\right] \tag{II.14}$$

$$= \frac{1}{8}\left(8 - 3 \times 2 - 7 \times 0\right) \tag{II.15}$$

$$= \frac{1}{8} \times 2 = \frac{1}{4} \tag{II.16}$$

Conclusion :

Nous avons $x_1^{(1)} = 2$, $x_2^{(1)} = 0$, $x_3^{(1)} = \frac{1}{4}$. Et donc, $x^{(1)} = \begin{pmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ \frac{1}{4} \end{pmatrix} 0$

Résolution par le calcul matriciel

Dans la section II.1.2, nous avons vu que l'on pouvait décomposer la matrice A par une matrice diagonale D, une matrice triangulaire inférieure E, et une matrice triangulaire supérieure F. Ceci fait, nous pouvons obtenir le vecteur x par la formule suivante :

$$x^{(k+1)} = D^{-1}[b - (E + F)x^{(k)}]$$

Nous avons alors :

$$A = \begin{pmatrix} 1 & 2 & 2 \\ 1 & 3 & 3 \\ 3 & 7 & 8 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 8 \end{pmatrix}}_{D} - \underbrace{\begin{pmatrix} 0 & -2 & -2 \\ 0 & 0 & -3 \\ 0 & 0 & 0 \end{pmatrix}}_{E} - \underbrace{\begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 0 \\ -3 & -7 & 0 \end{pmatrix}}_{F}$$

Nous obtenons alors :

$$x^{(1)} = \begin{pmatrix} \frac{1}{1} & 0 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{8} \end{pmatrix} \left[\begin{pmatrix} 2 \\ 2 \\ 8 \end{pmatrix} - \left(\begin{pmatrix} 0 & -2 & -2 \\ 0 & 0 & -3 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 0 \\ -3 & -7 & 0 \end{pmatrix} \right) \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right] \tag{II.17}$$

$$= \begin{pmatrix} \frac{1}{1} & 0 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{8} \end{pmatrix} \begin{pmatrix} 2 \\ 2 \\ 8 \end{pmatrix} \tag{II.18}$$

$$= \begin{pmatrix} 2 \\ \frac{2}{3} \\ 1 \end{pmatrix} \tag{II.19}$$

Remarque : Au fur et à mesure des itérations, le vecteur x donné par le calcul itératif effectué dans la partie II.1.4 se rapproche de la solution donnée par le précédent calcul. Il est alors normal que le vecteur trouvé au bout de la première itération soit différent du vecteur trouvé ci-dessus.

II.1.5 Implémentation

Commentaires fonctionnels

Note : L'implémentation qui suit utilise exactement les mêmes fonctions usuelles de manipulation de matrice que l'implémentation de l'algorithme de Gauss décrit dans la section I.3.1. De plus, dans cette implémentation, nous définirons une variable k qui sera notre compteur d'itération et qui permettra l'arrêt de notre code si la suite ne converge pas. Enfin, notre variable erreur ε sera mise à jour à chaque itération de la manière suivante :

$$\varepsilon^{(k)} = p^{(k)} = Max_{i=1,...,n}|\overline{x}_i - \widetilde{x}_i^k|$$

Nous détaillerons dans cette section uniquement les fonctions dites "non-usuelles" qui vont nous servir pour l'implémentation de l'algorithme de Gauss-Seidel. Il s'agit ici de la fonction de mise à jour de notre variable erreur et de la fonction implémentant l'algorithme de Gauss-Seidel.

Fonction *majEpsilon :*

La fonction **majEpsilon** permet de mettre à jour la variable d'erreur ε lors de l'exécution de notre algorithme. Grâce au vecteur $x^{(k)}$ qui représente la solution actuelle de notre système d'équations, la fonction calcule la différence absolue entre chaque élément $x_i^{(k)}$ et 1. Cela permet de mesurer à quel point les valeurs actuelles se rapprochent de 1, qui est notre valeur cible pour les solutions convergentes. La fonction conserve le maximum de ces différences absolues en tant que mesure d'erreur afin de mettre à jour notre variable ε. Cela permet de contrôler la précision de l'algorithme et de décider quand il a convergé de manière satisfaisante vers la solution recherchée. La fonction **majEpsilon** joue donc un rôle dans la détermination du critère d'arrêt de l'algorithme. Voici son implémentation en C :

```c
float majEpsilon(float** matXk, int row){
  float maxforEps=0;
  for(int i=0; i<row; i++){
      float soustr=fabs(1-matXk[i][0]);
      printf("%f\n", matXk[i][0]);
      if (soustr>maxforEps){
          maxforEps=soustr;
      }
  }
  return maxforEps;
}
```

Fonction *gaussSeidel :*

La fonction **gaussSeidel** implémente l'algorithme de Gauss-Seidel tel que décrit précédemment dans la section II.1.3. Par la programmation itérative, notre algorithme mettra à jour les solutions actuelles jusqu'à ce que l'erreur minimale définie soit atteinte ou que le nombre maximal d'itérations soit atteint. Voici son implémentation en C :

```c
  float** gaussSeidel(float **matA, float **matB, float **matXk, int row, int column,
                      int nbIterMax)
  {
    //CREATING OUR SOLUTION VECTOR AT ITERATION k
    float **matXk1=createMatrix(row, 1);

    //INITIALIZING OUR ERROR VARIABLE AND ITERATION COUNTER
    float epsilon=majEpsilon(matXk, row);;
    int iter=0;

```

```
    while ((epsilon>=pow(10,-6)) && (iter<nbIterMax)){
        for(int i=0; i<row ; i++){
            float sumF=0;
            float sumE=0;

            //CALCULATION OF F
            for(int j=i+1; j<row ; j++){
                sumF+=matA[i][j]*matXk[j][0];
            }

            //CALCULATION OF E
            for(int j=0; j<i ; j++){
                sumE+=matA[i][j]*matXk1[j][0];
            }

            //CALCULATION OF ELEMENT X_i^(k)
            matXk1[i][0]=(matB[i][0]-sumF-sumE)/matA[i][i];

        }

        //UPDATING OUR SOLUTION VECTOR
        for(int k=0; k<row ; k++){
            matXk[k][0]=matXk1[k][0];
        }

        //UPDATING OUR ERROR VARIABLE AND ITERATION COUNTER
        epsilon=majEpsilon(matXk, row);
        iter+=1;
    }

    //RETURN THE SOLUTION VECTOR
    return matXk;
}
```

II.1.6 Exemples d'exécution

Soient les matrices A données dans le TP et dans l'annexe du document (section **??**). En résolvant le système $Ax = b$ en question, nous obtenons respectivement les résultats suivants :

Listing II.1 – $A_1X = B$ results

```
                A matrix
 3.000   0.000   4.000
 7.000   4.000   2.000
 -1.000   1.000   2.000
                B matrix
 7.000
 13.000
 2.000
                SOLVING
                SOLUTION VECTOR X
-nan
-nan
-nan
Temps d'execution : 0.000380 secondes
```

Listing II.2 – $A_2X = B$ results

```
                A matrix
 -3.000   3.000   -6.000
 -4.000   7.000   8.000
 5.000   7.000   -9.000
```

```
                      B matrix
 -6.000
 11.000
 3.000
                      SOLVING
                      SOLUTION VECTOR X
-nan
-nan
-nan
Temps d'execution : 0.000328 secondes
```

Listing II.3 – $A_3X = B$ results

```
                      A matrix
 4.000   1.000   1.000
 2.000   -9.000   0.000
 0.000   -8.000   6.000
                      B matrix
 6.000
 -7.000
 -2.000
                      SOLVING
                      SOLUTION VECTOR X
 1.000
 1.000
 1.000
Temps d'execution : 0.000257 secondes
```

Listing II.4 – $A_4X = B$ results

```
                      A matrix
 7.000   6.000   9.000
 4.000   5.000   -4.000
 -7.000   -3.000   8.000
                      B matrix
 22.000
 5.000
 -2.000
                      SOLVING
                      SOLUTION VECTOR X
 1.000
 1.000
 1.000
Temps d'execution : 0.000267 secondes
```

Listing II.5 – $A_5X = B$ results

```
                      A matrix
 1.000   0.500   0.250
 0.500   1.000   0.000
 0.250   0.000   1.000
                      B matrix
 1.750
 1.500
 1.250
                      SOLVING
                      SOLUTION VECTOR X
 1.000
 1.000
 1.000
Temps d'execution : 0.000286 secondes
```

Listing II.6 – $A_6X = B$ results

```
                    A matrix
  1.000   0.500   0.250   0.125   0.062   0.031
  0.500   1.000   0.000   0.000   0.000   0.000
  0.250   0.000   1.000   0.000   0.000   0.000
  0.125   0.000   0.000   1.000   0.000   0.000
  0.062   0.000   0.000   0.000   1.000   0.000
  0.031   0.000   0.000   0.000   0.000   1.000
                    B matrix
  1.969
  1.500
  1.250
  1.125
  1.062
  1.031
                    SOLVING
                    SOLUTION VECTOR X
  1.000
  1.000
  1.000
  1.000
  1.000
  1.000
  Temps d'execution : 0.000305 secondes
```

Listing II.7 – $A_7X = B$ results

```
                    A matrix
  1.000   0.500   0.250   0.125   0.062   0.031   0.016   0.008
  0.500   1.000   0.000   0.000   0.000   0.000   0.000   0.000
  0.250   0.000   1.000   0.000   0.000   0.000   0.000   0.000
  0.125   0.000   0.000   1.000   0.000   0.000   0.000   0.000
  0.062   0.000   0.000   0.000   1.000   0.000   0.000   0.000
  0.031   0.000   0.000   0.000   0.000   1.000   0.000   0.000
  0.016   0.000   0.000   0.000   0.000   0.000   1.000   0.000
  0.008   0.000   0.000   0.000   0.000   0.000   0.000   1.000
                    B matrix
  1.992
  1.500
  1.250
  1.125
  1.062
  1.031
  1.016
  1.008
                    SOLVING
                    SOLUTION VECTOR X
  1.000
  1.000
  1.000
  1.000
  1.000
  1.000
  1.000
  1.000
  Temps d'execution : 0.000369 secondes
```

Listing II.8 – $A_8X = B$ results

```
                    A matrix
  1.000   0.500   0.250   0.125   0.062   0.031   0.016   0.008   0.004   0.002
  0.500   1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
  0.250   0.000   1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
  0.125   0.000   0.000   1.000   0.000   0.000   0.000   0.000   0.000   0.000
  0.062   0.000   0.000   0.000   1.000   0.000   0.000   0.000   0.000   0.000
  0.031   0.000   0.000   0.000   0.000   1.000   0.000   0.000   0.000   0.000
  0.016   0.000   0.000   0.000   0.000   0.000   1.000   0.000   0.000   0.000
  0.008   0.000   0.000   0.000   0.000   0.000   0.000   1.000   0.000   0.000
  0.004   0.000   0.000   0.000   0.000   0.000   0.000   0.000   1.000   0.000
  0.002   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   1.000
                    B matrix
  1.998
  1.500
```

```
1.250
1.125
1.062
1.031
1.016
1.008
1.004
1.002
                    SOLVING
                    SOLUTION VECTOR X
1.000
1.000
1.000
1.000
1.000
1.000
1.000
1.000
1.000
1.000
Temps d'execution : 0.000351 secondes
```

Listing II.9 – $A_9X = B$ results

```
                    A matrix
3.000     -1.000    0.000
-2.000    3.000     -1.000
0.000     -2.000    3.000
                    B matrix
2.000
0.000
1.000
                    SOLVING
                    SOLUTION VECTOR X
1.000
1.000
1.000
Temps d'execution : 0.000274 secondes
```

Listing II.10 – $A_10X = B$ results

```
                    A matrix
3.000     -1.000    0.000     0.000     0.000     0.000
-2.000    3.000     -1.000    0.000     0.000     0.000
0.000     -2.000    3.000     -1.000    0.000     0.000
0.000     0.000     -2.000    3.000     -1.000    0.000
0.000     0.000     0.000     -2.000    3.000     -1.000
0.000     0.000     0.000     0.000     -2.000    3.000
                    B matrix
2.000
0.000
0.000
0.000
0.000
1.000
                    SOLVING
                    SOLUTION VECTOR X
1.000
1.000
1.000
1.000
1.000
1.000
Temps d'execution : 0.000315 secondes
```

Listing II.11 – $A_11X = B$ results

```
                    A matrix
```

```
3.000   -1.000   0.000   0.000   0.000   0.000   0.000   0.000
-2.000   3.000   -1.000   0.000   0.000   0.000   0.000   0.000
0.000   -2.000   3.000   -1.000   0.000   0.000   0.000   0.000
0.000   0.000   -2.000   3.000   -1.000   0.000   0.000   0.000
0.000   0.000   0.000   -2.000   3.000   -1.000   0.000   0.000
0.000   0.000   0.000   0.000   -2.000   3.000   -1.000   0.000
0.000   0.000   0.000   0.000   0.000   -2.000   3.000   -1.000
0.000   0.000   0.000   0.000   0.000   0.000   -2.000   3.000
                B matrix
2.000
0.000
0.000
0.000
0.000
0.000
0.000
1.000
                SOLVING
                SOLUTION VECTOR X
1.000
1.000
1.000
1.000
1.000
1.000
1.000
1.000
Temps d'execution : 0.000341 secondes
```

Listing II.12 – $A_12X = B$ results

```
                A matrix
3.000   -1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
-2.000   3.000   -1.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   -2.000   3.000   -1.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   -2.000   3.000   -1.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   -2.000   3.000   -1.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   -2.000   3.000   -1.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   -2.000   3.000   -1.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   -2.000   3.000   -1.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   -2.000   3.000   -1.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   -2.000   3.000
                B matrix
2.000
0.000
0.000
0.000
0.000
0.000
0.000
0.000
0.000
1.000
                SOLVING
                SOLUTION VECTOR X
1.000
1.000
1.000
1.000
1.000
1.000
1.000
1.000
1.000
1.000
Temps d'execution : 0.000479 secondes
```

Chapitre III

Méthode de Jacobi

III.1 Méthode de Jacobi

Rappelons que la méthode de **Jacobi** est itérative et ne garantit pas toujours un résultat. La méthode est définie si A est définie positive.
L'algorithme permet de trouver un résultat si la matrice est dites à diagonale strictement dominante. Autrement dit, Soit $[a_{ij}]_{0\leq i,j\leq n}$ les coefficients réels peuplant $A \in \mathcal{M}_{n,n}(\mathbb{R})$, alors si :
$\forall i, |a_{i,i}| < \sum\limits_{i\neq j} |a_{ij}|$, on a que Jacobi converge vers l'unique solution du système $Ax = b$.

III.1.1 Principe de la méthode

On veut résoudre $Ax = b$ avec $A \in \mathcal{M}_{n,n}(\mathbb{R})$, $n \in \mathbb{N}$, x la vecteur colonne contenant les inconnus et b le vecteur colonne des solution.
On pose $D \in \mathcal{M}_{n,n}(\mathbb{R})$ la matrice contenant les coefficients $[a_{i,j}]_{0\leq i=j\leq n}$ de A.
On pose aussi E et F avec E la matrice triangulaire opposée inférieure de A et F la matrice supérieure opposée de A.
On obtient alors :

$$Ax = b \tag{III.1}$$
$$(D - E - F)x = b \tag{III.2}$$
$$Dx - (E + F)x = b \tag{III.3}$$
$$x = D^{-1}(E + F)x + D^{-1}b \tag{III.4}$$
$$x^{k+1} = D^{-1}(E + F)x^k + D^{-1}b \tag{III.5}$$

Ce qui donne l'algorithme suivant :
Soit ϵ L'erreur maximale, un point initial x^0 et $k = 0$
avec $\epsilon^0 = ||Ax^0 - b||$
On obtient :

Tant que $(\epsilon^{(k)} \leq \epsilon)$
$x_i^{k+1} = \frac{1}{a_{ii}}[b_i - \sum\limits_{j\neq i} a_{ij}x_j^{(k)}], i = 1, \ldots, n$
$\epsilon^{k+1} = ||Ax^{k+1} - b||$
$k = k + 1$
FIN JACOBI

Remarque, on ajoutera aussi un nombre d'iterations maximum afin de ne pas être dans le cas d'une boucle infinies (si jacobi diverge alors l'erreur augmente).

III.1.2 Résolution manuelle

Nous en détaillerons seulement une itération

Soit $A = \begin{pmatrix} 4 & 1 & 0 \\ -1 & 3 & 6 \\ -2 & -5 & -3 \end{pmatrix}$, $b = \begin{pmatrix} 8 \\ 3 \\ 8 \end{pmatrix}$,$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$ et $x^{(0)} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$

On a $A = \underbrace{\begin{pmatrix} 4 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & -2 \end{pmatrix}}_{D} - \underbrace{\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 2 & 5 & 0 \end{pmatrix}}_{E} - \underbrace{\begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & -6 \\ 0 & 0 & 0 \end{pmatrix}}_{F}$

On a donc $x^{k+1} = D^{-1}[(E+F)x^k + b]$

Dans le cas présent on obtient alors :

$x^1 = \begin{pmatrix} \frac{1}{4} & 0 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & 0 & -\frac{1}{2} \end{pmatrix} \left[\left[\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 2 & 5 & 0 \end{pmatrix} + \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & -6 \\ 0 & 0 & 0 \end{pmatrix} \right] \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 8 \\ 3 \\ 8 \end{pmatrix} \right]$

$x^1 = \begin{pmatrix} 2 \\ 1 \\ -4 \end{pmatrix}$

et

$$\begin{aligned}
\epsilon^{(1)} &= ||Ax^{(1)} - b|| \\
\epsilon^{(1)} &= || \begin{pmatrix} 4 & 1 & 0 \\ -1 & 3 & 6 \\ -2 & -5 & -3 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ -4 \end{pmatrix} - \begin{pmatrix} 8 \\ 3 \\ 8 \end{pmatrix} || \\
\epsilon^{(1)} &= || \begin{pmatrix} 1 \\ -26 \\ -5 \end{pmatrix} || \\
\epsilon^{(1)} &= \sqrt{1^2 + (-26)^2 + (-5)^2} \\
\epsilon^{(1)} &= \sqrt{(702)}
\end{aligned}$$

III.1.3 Implémentation

Pour l'implémentation de cette méthode, nous utiliserons ϵ comme suit :

$$\epsilon^{(k)} = p^k = \text{Max}_{i=1,\dots,n} |\bar{x_i} - \tilde{x_i}^k|$$

Où $\bar{x_i}$ est les résultat attendu et $\tilde{x_i}^k$ est l'approximation trouvée à l'étape k.
De plus on utilisera aussi une limite d'occurrence, pour pouvoir gérer les matrices où **Jacobi** diverge.
En l'occurrence on se fixera un $\epsilon = 10^{-6}$ et un nombre d'itération maximum fixé à 1000

III.1.4 Code

On ne détaillera ici seulement les fonctions dites "non triviales" c'est-à-dire les fonctions étant en lien directe avec l'algorithme décrit.
De plus chaque fonction présentée sera dépouillée de toute fonction d'affichage permettant de produire des chiffres liés à l'utilisation du programme (pour une question de lisibilité).

Listing III.1 – jacobi.c

```
int jacobi(float **A, float *vector, float *b, float *S, int n, float minErr,int bound) {
  float epsilon = epsi(S, vector, n);
  int k = 0;
  float *cp = malloc(n * sizeof *cp);
  while (k < bound && epsilon >= minErr) {
```

```
    fprintf(stderr, "%f ", epsilon);
    // printf("EPSILON : %f\n", epsilon);
    copy(cp, vector, n);
    for (int i = 0; i < n; i++) {
      vector[i] = (1 / A[i][i]) * (b[i] - jacobiSum(A, cp, n, i));
    }
    epsilon = epsi(vector, S, n);
    k++;
  }
  free(cp);
  return k;
}
```

Commentaires : Nous remarquerons que l'implémentation de l'algorithme de Jacobi est similaire à ce que a été présenté ultérieurement. Pour des questions de lisibilité, $\epsilon^{(k)}$ est calculée par la fonction *epsi()*, de même pour **jacobiSum()**, le calcul a été séparé afin de faciliter la compréhension du programme.

Listing III.2 – jacobiSum() function in "source.h"

```
float jacobiSum(float **A, float *V, int m, int i) {
  float s = 0;
  for (int j = 0; j < m; j++) {
    if (j != i)
      s += A[i][j] * V[j];
  }
  return s;
}
```

Commentaire : Permet de calculer de façon lisible est clair ceci :

$$\sum_{j\neq i} a_{ij}x_j^{(k)}, i = 1, \dots k$$

Listing III.3 – epsi() function in "source.h"

```
float epsi(float *V, float *S, int n) {
  float max = Fabs(V[0] - S[0]);
  for (int i = 1; i < n; i++) {
    if (Fabs(S[i] - V[i]) > max)
      max = Fabs(S[i] - V[i]);
  }
  return max;
}
```

Commentaires : Utilise la fonction *Fabs()* que nous avons implémenter, elle renvoie la valeur absolue d'un nombre flottant.
Cette fonction *epsi()* permet donc de calculer l'erreur entre deux itération de la façon suivante :

$$\epsilon^{(k)} = \text{Max}_{i=1,\dots,n}|\bar{x_i} - \tilde{x_i}^k|$$

Listing III.4 – conv() function in "source.h"

```
int conv(float **A, int m) {
  float sl = 0;
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < m; j++) {
      if (i != j)
        sl += fabs(A[i][j]);
    }
    if (A[i][i] - sl <= 0)
      return 0;
  }
```

```
    return 1;
}
```

Commentaire *conv()* est une fonction utilitaire permettant d'effectuer une prediction sur la convergeance potentielle d'une matrice par jacobi.
Pour cela on vérifie si la matrice sur laquelle on effectue jacobi est à diagonale strictemennt dominante.
Ce qui se vérifie par cette formule :

$$\forall i, |a_{ii}| > \sum_{j \neq i} |a_{ij}|$$

III.1.5 Entrées / Sorties

Entrées

Le programme prend en entrée 2 paramètres soient

1. minErr, ou l'erreur minimale tolérée
2. bounds, qui désigne la limite d'occurrence du programme (en cas de divergeance)

L'utilisateur peut ensuite décider de remplir manuellement sa matrice où de rediriger le flux d'un fichier comme suit :

Listing III.5 – A4.txt

```
3 3
7 6 9
4 5 −4
−7 −3 8
```

Avec sur la première ligne : les dimensions de la matrice suivit, sur les ligne suivantes, des coefficients de la matrice

III.1.6 Sorties

Le flux d'erreur sera réservé afin de produire des données engendrant des graphiques (des données formatées). Il s'agit de l'énumération de tout les ϵ calculés suivit du nombre d'itération.
Sur les autre lignes seront inscrit des messages facilitant la prise en main du programme pour l'utilisateur.
Il figuera ensuite la prédiction quant à la convergence de la méthode.
Enfin la dernière lignes représentera le nombre de ϵ calculés.

III.1.7 Exécution

Voici l'exécution sur les 12 matrices de la méthode de jacobi avec $\epsilon = 10^{-4}$.
La redirection de flux de sortie du programme est la suivante : >

Listing III.6 – Execution with A1 matrix

```
A matrix dimensions:
Enter coefficient for 0x55ce91727300[0][0]Enter coefficient for 0x55ce91727300[0][1]Enter coefficient fo
 ====== CONVERGENCE PREDICTION: may not conv ======
PRINTING VECTOR FROM: 0x55ce917272e0 LOCATION :
−nan −48517597648316769037382673682594267136.000000 inf
EPSILON CALCULATED 742
```

Listing III.7 – Execution with A2 matrix

```
A matrix dimensions:
Enter coefficient for 0x55ea0288b300[0][0]Enter coefficient for 0x55ea0288b300[0][1]Enter coefficient fo
```

```
 ====== CONVERGENCE PREDICTION: may not conv ======
PRINTING VECTOR FROM: 0x55ea0288b2e0 LOCATION :
-nan -nan -inf
EPSILON CALCULATED 252
```

Listing III.8 – Execution with A3 matrix

```
A matrix dimensions:
 Enter coefficient for 0x5579594bf300[0][0]Enter coefficient for 0x5579594bf300[0][1]Enter coefficient fo
 ====== CONVERGENCE PREDICTION: may not conv ======
PRINTING VECTOR FROM: 0x5579594bf2e0 LOCATION :
 1.000052 1.000013 0.999954
EPSILON CALCULATED 13
```

Listing III.9 – Execution with A4 matrix

```
A matrix dimensions:
 Enter coefficient for 0x560374865300[0][0]Enter coefficient for 0x560374865300[0][1]Enter coefficient fo
 ====== CONVERGENCE PREDICTION: may not conv ======
PRINTING VECTOR FROM: 0x5603748652e0 LOCATION :
 1.000078 0.999903 1.000078
EPSILON CALCULATED 24
```

Listing III.10 – Execution with A5 dimensions : 3x3

```
A matrix dimensions:
 Enter coefficient for 0x56131ea84300[0][0]Enter coefficient for 0x56131ea84300[0][1]Enter coefficient fo
 ====== CONVERGENCE PREDICTION: may not conv ======
PRINTING VECTOR FROM: 0x56131ea842e0 LOCATION :
 1.000068 1.000045 1.000023
EPSILON CALCULATED 17
```

Listing III.11 – Execution with A5 dimensions : 6x6

```
A matrix dimensions:
 Enter coefficient for 0x55b573d18300[0][0]Enter coefficient for 0x55b573d18300[0][1]Enter coefficient fo
 ====== CONVERGENCE PREDICTION: may not conv ======
PRINTING VECTOR FROM: 0x55b573d182e0 LOCATION :
 0.999950 0.999927 0.999963 0.999982 0.999991 0.999995
EPSILON CALCULATED 18
```

Listing III.12 – Execution with A5 dimensions : 8x8

```
A matrix dimensions:
 Enter coefficient for 0x56527ea95320[0][0]Enter coefficient for 0x56527ea95320[0][1]Enter coefficient fo
 ====== CONVERGENCE PREDICTION: may not conv ======
PRINTING VECTOR FROM: 0x56527ea952f0 LOCATION :
 0.999949 0.999924 0.999962 0.999981 0.999991 0.999995 0.999998 0.999999
EPSILON CALCULATED 18
```

Listing III.13 – Execution with A5 dimensions : 10x10

```
A matrix dimensions:
 Enter coefficient for 0x55cfc4470320[0][0]Enter coefficient for 0x55cfc4470320[0][1]Enter coefficient fo
 ====== CONVERGENCE PREDICTION: may not conv ======
PRINTING VECTOR FROM: 0x55cfc44702f0 LOCATION :
 0.999949 0.999924 0.999962 0.999981 0.999990 0.999995 0.999998 0.999999 0.999999 1.000000
EPSILON CALCULATED 18
```

Listing III.14 – Execution with A6 dimensions : 3x3

```
A matrix dimensions:
```

```
Enter coefficient for 0x55774b4d7300[0][0]Enter coefficient for 0x55774b4d7300[0][1]Enter coefficient fo
 ===== CONVERGENCE PREDICTION: may not conv =====
PRINTING VECTOR FROM: 0x55774b4d72e0 LOCATION :
0.999956 0.999941 0.999911
EPSILON CALCULATED 24
```

Listing III.15 – Execution with A6 dimensions : 6x6

```
A matrix dimensions:
Enter coefficient for 0x562479ac8300[0][0]Enter coefficient for 0x562479ac8300[0][1]Enter coefficient fo
 ===== CONVERGENCE PREDICTION: may not conv =====
PRINTING VECTOR FROM: 0x562479ac82e0 LOCATION :
0.999989 0.999967 0.999949 0.999917 0.999917 0.999926
EPSILON CALCULATED 61
```

Listing III.16 – Execution with A6 dimensions : 8x8

```
A matrix dimensions:
Enter coefficient for 0x55a7d738a320[0][0]Enter coefficient for 0x55a7d738a320[0][1]Enter coefficient fo
 ===== CONVERGENCE PREDICTION: may not conv =====
PRINTING VECTOR FROM: 0x55a7d738a2f0 LOCATION :
0.999994 0.999985 0.999968 0.999954 0.999927 0.999918 0.999904 0.999936
EPSILON CALCULATED 84
```

Listing III.17 – Execution with A6 dimensions : 10x10

```
A matrix dimensions:
Enter coefficient for 0x55cc8327d320[0][0]Enter coefficient for 0x55cc8327d320[0][1]Enter coefficient fo
 ===== CONVERGENCE PREDICTION: may not conv =====
PRINTING VECTOR FROM: 0x55cc8327d2f0 LOCATION :
0.999997 0.999992 0.999983 0.999974 0.999955 0.999942 0.999918 0.999912 0.999902 0.999934
EPSILON CALCULATED 104
```

III.1.8 Remarque sur les résultats

On remarquera que lorsque Jacobi renvoie **NaN, inf ou des valeurs proches de la limite d'encodage du type int (4 octets)** alors il s'agit du cas où cette méthode diverge, donc aucun résultat fiable n'est envisageable.
On rappelera aussi que dans le cadre de ce programme, si la méthode renvoie des valeurs **extrement proches** de 1, alors le programme a trouvé une solution. report [french]babel [T1]fontenc [utf8]inputenc mathtools amssymb hyperref float amsthm listings geometry setspace graphicx fancyhdr subcaption cleveref

Définition

hmargin=3cm,vmargin=2.5cm

III.2 Conclusion Générale des Méthodes Itératives

III.2.1 Tableau récapitulatif

A_i	$p(J)$	NbIterJacobi	$p(GS)$	NbIterGauss-Seidel
A_1	∞	742	∞	192
A_2	∞	252	∞	128
A_3	0.000001	19	0.00000	9
A_4	0.000001	35	0.000001	59
A_5	0.000001	25	0.000001	14
A_6	0.000002	26	0.000001	15
A_7	0.000001	26	0.000001	15
A_8	0.000002	26	0.000001	15
A_9	0.000002	35	0.000001	19
A_{10}	0.000002	88	0.000001	44
A_{11}	0.000001	120	0.000001	60
A_{12}	0.000001	140	0.000001	73

III.2.2 Conclusion

Comme peuvent le démontrer les différents graphiques ainsi que le tableau ci-dessus, nous remarquerons que la Méthode de **Gauss-Seidel** reste majoritairement plus efficace que la méthode de **Jacobi**. Nous insisterons sur le fait que la Méthode de **Gauss-Seidel** est particulièrement adaptée pour le calcul parallèle alors que la méthode de **Jacobi** est plus adaptée sur des matrices creuses.