Rapport de projet Mathématiques.

15 décembre 2022

Sommaire

Table des matières

1 Introduction

1.1 Motivation

ds

1.2 Théorie

Le but est de résoudre l'équation linéaire de matrice : $AX = B$ dans $\mathbb{R}^N$ sans avoir à calculer A^{-1}. Car calculer A^{-1} peut poser problème lorsque N devient grand. En effet, pour $N = 20$, il faudra 5 fois l'age de l'univers pour calculer A^{-1}.

Nous allons donc à l'aide de plusieurs méthodes, tenter de simplifier le problème en sous-problèmes plus simple à résoudre et procéder à des optimisations graduelles afin d'optimiser le cout en temps de calcul et l'espace mémoire utilisé pour le cacul.

2 Références

2.1 Bibliographie

Références

2.2 Biographies

2.2.1 Gauss

Avant d'étudier la méthode de Gauss, il peut être pertinent d'établir une courte biographie de son auteur. Surnommé "prince des mathématiques" par ses pairs, ce mathématicien, né le 30 avril 1777, a contribué de bien des manières à développer non seulement les mathématiques mais également les méthodes en astrophysique et en électromagnétisme. Johann Carl Friedrich Gauss a pu se pencher sur des problèmes dits classiques (depuis l'antiquité) en adoptant des méthodes et raisonnements modernes. Il démontre alors le théorème fondamental de l'algèbre que l'on connaît aujourd'hui sous le nom du théorème de d'Alembert-Gauss, il dédie également un ouvrage à la théorie des nombres contenant plusieurs démonstrations qui révolutionnent l'arithmétique, on lui doit en partie la forme actuelle des nombres complexes. En astrophysique, il met au point la méthodes des moindres carrés permettant de minimiser les incertitudes dues aux mesures ce qui lui permet de déterminer exactement la position de Ceres, une planète naine du système solaire. Avec la contribution de Wilhem Weber, Gauss formule deux théorèmes essentiels en électromagnétisme réfutant l'existence de monopôle magnétique et établissant une relation entre le flux d'un champ électrique sur une surface fermée et la charge électrique totale à l'intérieur de cette surface. Cette liste est loin de résumer tous ses travaux, dont une partie a été publiée après son décès le 23 février 1855, ces derniers sont très nombreux et ont servi de base de recherche pour d'autres mathématiciens et physiciens après lui. Il est donc intéressant d'étudier la solution apportée par Gauss et de l'appliquer à notre problématique. L'élimination de Gauss est expliquée ci-après.

2.2.2 Cholesky

André-Louis Cholesky, également appelé René Cholesky, est un polytechnicien français qui s'est engagé dans l'armée suite à sa formation. Au cours de sa carrière militaire, on lui assigne différentes missions qui l'emmènent à plusieurs endroits où il effectue notamment des travaux de triangulation. Il participe à la Première Guerre mondiale où il est blessé et il meurt le 31 août 1918 des suites de ses blessures. Cependant, on le connaît plus pour ses conctributions aux mathématiques que pour sa carrière de militaire. En effet, il est l'auteur d'un manuscrit intitulé "Sur la résolution numérique des systèmes d'équations linéaires". Cette méthode est en fait une nouvelle approche de la méthode des moindres carrés et sera publiée 6 ans après sa mort.

3 Travail préliminaire

3.1 Type de donnée

Tout d'abord, en C, comme la plupart des langages de programmation, il est impossible d'exprimer l'intégralité des réels, entiers, complexes ... car il faudrait disposer d'une mémoire infinie pour représenter une infinité de nombres. Comme nous utilisons des nombres réels pour les vecteurs et matrices, il faut choisir parmis les types signés flottant du C qui sont :

Type	Taille	Portée	Précision maximale
float	32 bits / 4 octets	$1,2 \cdot 10^{-38}$ à $3,4 \cdot 10^{38}$	6 chiffre après la virgule
double	64 bits / 8 octets	$2,3 \cdot 10^{-308}$ à $1,7 \cdot 10^{308}$	15 chiffres après la virgule
long double	80 bits / 10 octets	$3,4 \cdot 10^{-4932}$ à $1,1 \cdot 10^{4932}$	19 chiffres après la virgule

Dans ce projet, le point qui nous intéresse le plus est celui de la performance de la résolution, cette dernière étant très exigeante en temps processeur. Ainsi, le choix du type est crucial. Au final, nous avons choisi le type "double" car il offre une précision et une portée amplement suffisante, sans compromettre les performances de nos algorithmes. En effet, contrairement au type « long double », le type « double » nécessite pas de temps processeur supplémentaire par rapport à un type « float » sur les ordinateurs modernes. C'est à dire, dont les processeur possèdent des registres de calculs flottant de 64 bits / 8 octets. Il est à noter que cette considération aurait été différente il y 10 - 20 ans car les processeurs d'antan ne supportaient ne possédaient que des registres de calcul flottant de 32 bits, 16 bits voir 8 bits (pour les plus anciens). De plus, bien que le type « double » occupe deux fois plus de mémoire que le type « float », ici l'impact sur la mémoire reste relativement limité comparé à la précision supplémentaire offerte.

3.2 Implémentation des matrices et vecteurs

Ensuite, les concepts de matrice et de vecteur en C ne sont pas directement implémenté dans le langage. Cependant, nous pouvons utiliser une liste pour les vecteurs et des listes imbriqués dans des listes (tableau à 2 dimension) pour les matrices.

Ainsi, pour l'allocation des vecteurs et des matrices, nous avons implémenté les fonctions suivantes :

```
double *Allocation_Vecteur(int Taille)
{
    // Allocation d'un espace de mémoire de taille =
       Taille Vecteur * Taille du type (8 octets).
    double * Vecteur = (double *)malloc(Taille *
       sizeof(double));

    // Nettoyage du vecteur.
    Nettoyage_Vecteur(Vecteur, Taille);
```

```
    return Vecteur;
}

double **Allocation_Matrice(int Taille)
{
    // Allocation de la première dimension
    double **Matrice = (double **)malloc(Taille *
        sizeof(double));

    // Allocation de la deuxième dimension
    for (int i = 0; i < Taille; i++)
    {
        Matrice[i] = Allocation_Vecteur(Taille);
    }

    // Nettoyage de la matrice (remplissage par des 0)
    Nettoyage_Matrice(Matrice, Taille);

    return Matrice;
}
```

Ces fonctions appellent les fonctions “Nettoyage_Vecteur” et “Nettoyage_Matrice”. Ces fonctions viennent nettoyer les matrices et vecteurs en les remplissant par des 0 car la mémoire alloué en C peut ne pas être toujours propre (contenir des valeurs aléatoires) :

```
void Nettoyage_Vecteur(double *Vecteur, int Taille)
{
    // Itère parmis les éléments du vecteur
    for (int i = 0; i < Taille; i++)
    {
        // Remplissage par des 0
        Vecteur[i] = 0;
    }
}

void Nettoyer_Matrice(double **A, int Taille)
{
    // Itère parmis la première dimension
    for (int i = 0; i < Taille; i++)
    {
        // Nettoyage de la deuxième dimension
        Nettoyage_Vecteur(A[i], Taille);
    }
}
```

Ensuite, afin de rendre l’affichage des vecteurs et matrices lisible, nous avons implémenté les fonctions suivantes :

```c
void Afficher_Vecteur(double *Vecteur, int Taille)
{
    // Saut de ligne.
    printf("\n");
    // Itère parmis les éléments du vecteur.
    for (int i = 0; i < Taille; i++)
    {
        // Affichage de la valeur suivi d'un saut de
           ligne.
        printf("%f\n", Vecteur[i]);
    }
}

void Afficher_Matrice(double **Matrice, int Taille)
{
    // Itère parmis la première dimension de la
       matrice.
    for (int i = 0; i < Taille; i++)
    {
        // Retour à la ligne.
        printf("\n");

        // Itère parmis la deuxième dimension de la
           matrice.
        for (int j = 0; j < Taille; j++)
        {
            // Affichage de la valeur avec un
               séparateur.
            printf("| %f ", Matrice[i][j]);
        }
        // Retour à la ligne.
        printf("|\n");
    }
}
```

Enfin, comme nous utilisons de l'allocation dynamique pour les vecteurs et matrices, contrairement à l'allocation statique, la libération de la mémoire (désallocation) doit être faite manuellement :

4 Méthodes directes

4.1 Méthode triangulaire inférieure

Introduction

Soit B et X, des vecteurs de taille N et A une matrice triangulaire inférieure carrée de taille N. On cherche, pour A et B donné, X tel que :

$$AX = B$$

Méthode

Soit $N = 3$, on a donc :

$$AX = B$$

$$\Rightarrow \begin{pmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

$$\Rightarrow \begin{pmatrix} a_{11}x_1 \\ a_{21}x_1 + a_{22}x_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

$$\Rightarrow \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} \frac{b_1}{a_{11}} \\ \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}}x_1 \\ \frac{b_3}{a_{33}} - \frac{a_{31}}{a_{33}}x_1 - \frac{a_{32}}{a_{33}}x_2 \end{pmatrix}$$

On en déduit alors une formule générale pour X :

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j}{a_{ii}}, i = 1, \ldots, N.$$

Code

```
double *Sol_Inf(double **a, double *b, int Taille)
{
    // Allocation du vecteur X.
    double *x = Allocation_Vecteur(Taille);

    // Calcul du premier terme de X.
    x[0] = b[0] / a[0][0];

    // Itère parmis les lignes de la matrice.
    for (int i = 1; i < Taille; i++)
    {
```

```
            // Calcul de la somme des a[i][j] * x[j]
            double Sum = 0;
            for (int j = 0; j < i; j++)
            {
                Sum = Sum + a[i][j] * x[j];
            }
            // Calcul du terme X[i].
            x[i] = (b[i] - Sum) / a[i][i];
        }

        return x;
    }
```

Conclusion

Ainsi, cette algorithme possède une complexité temporelle maximale quadratique ($O\left(N^2\right)$) car nous avons deux boucles imbriqués les unes dans les autres qui dépendent plus ou moins de la taille N.

4.2 Méthode triangulaire supérieure

Introduction

Soit B un vecteur de taille N, A une matrice triangulaire supérieure carrée de taille N et X un vecteur de taille N tel. Comme pour la section précédente, on cherche pour A et B donné, X tel que :

$$AX = B$$

Exemple

Soit $N = 3$, on a :

$$AX = B$$

$$\Rightarrow \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \Leftrightarrow$$

$$\Rightarrow \begin{cases} a_{33}x_3 = b_3 \\ a_{22}x_2 + a_{23}x_3 = b_2 \\ a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \end{cases}$$

$$\Rightarrow \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} \frac{b_1}{a_{11}} - \frac{a_{12}}{a_{11}}x_2 - \frac{a_{13}}{a_{11}}x_3 \\ \frac{b_2}{a_{22}} - \frac{a_{23}}{a_{22}}x_3 \\ \frac{b_3}{a_{33}} \end{pmatrix} = \begin{pmatrix} \frac{b_1}{a_{11}} \\ \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}}x_1 \\ \frac{b_3 - \sum_{k=1}^{2} a_{3k}x_k}{a_{33}} \end{pmatrix}$$

On en déduit alors une formule générale pour X :

$$x_i = \frac{b_i - \sum_{j=i+1}^{N} a_{ij} x_j}{a_{ii}}, i = N, \ldots, 1$$

Code

```
Ecrire algorithme ici
```

Conclusion

De manière similaire à la précédente, cette algorithme possèdent une complexité quadratique ($O\left(N^2\right)$).

4.3 Élimination de Gauss

Introduction

Maintenant que nous disposons des fonctions pour résoudre $AX = B$ avec A une matrice triangulaire inférieure ou supérieure carrée. On va maintenant chercher à résoudre la même équation mais lorsque A est une matrice carrée quelconque. Nous allons utiliser la méthode de Gauss pour transformer l'expression $AX = B$ où A est une matrice carrée en $UX = e$ avec U qui est une matrice triangulaire supérieure. Ainsi, il ne restera plusq qu'a résoudre $UX = e$ avec la fonction « Sol_Sup » pour obtenir X.

Méthode

Soit $N = 3$, tel que :

$$\Rightarrow \underbrace{\begin{pmatrix} 3 & 1 & 2 \\ 3 & 2 & 6 \\ 6 & 1 & 1 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{X} = \underbrace{\begin{pmatrix} 2 \\ 1 \\ 4 \end{pmatrix}}_{B} \Leftrightarrow \underbrace{\begin{pmatrix} a & b & c \\ 0 & d & f \\ 0 & 0 & g \end{pmatrix}}_{U} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{X} = \underbrace{\begin{pmatrix} e_1 \\ e_2 \\ e_3 \end{pmatrix}}_{e}$$

On a donc :

$$\begin{matrix} (1) \\ (2) \\ (3) \end{matrix} \begin{pmatrix} 3x_1 + x_2 + 2x_3 \\ 3x_1 + 2x_2 + 6x_3 \\ 6x_1 + x_2 - x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ 4 \end{pmatrix} \Leftrightarrow \begin{pmatrix} ax_1 + bx_2 + cx_3 \\ dx_2 + fx_3 \\ gx_3 \end{pmatrix} = \begin{pmatrix} e_1 \\ e_2 \\ e_3 \end{pmatrix}$$

On va maintenant éliminer x_1 des équations (2) et (3) en utilisant (1) :

$$\begin{matrix} (1) \\ (2') \\ (3') \end{matrix} \begin{pmatrix} (1) \\ (2) - \frac{3}{3}(1) \\ (3) - \frac{3}{6}(1) \end{pmatrix} = \begin{pmatrix} 3x_1 + x_2 + 2x_3 \\ x_2 + 4x_3 \\ -x_2 - 5x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \\ 0 \end{pmatrix}$$

Puis, on va éliminer x_2 de l'équation $(3')$ en utilisant $(2')$:

$$\begin{matrix}(1)\\(2')\\(3'')\end{matrix}\begin{pmatrix}(1)\\(2)-\frac{3}{3}(1)\\(3')-\frac{-1}{1}(2')\end{pmatrix}=\begin{pmatrix}3x_1+x_2+2x_3\\x_2+4x_3\\-x_3\end{pmatrix}=\begin{pmatrix}2\\-1\\-1\end{pmatrix}$$

Si on factorise par X, on a donc bien U une matrice triangulaire supérieure carrée :

$$\underbrace{\begin{pmatrix}3&1&2\\0&1&4\\0&0&-1\end{pmatrix}}_{U}\underbrace{\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix}}_{X}=\underbrace{\begin{pmatrix}2\\-1\\-1\end{pmatrix}}_{e}$$

Il ne reste plus qu'à utiliser « Sol_Sup » pour résoudre cette équation.

Code

```
Code de mes fesses
void KILL_ME()
```

Conclusion

Cette méthode est de complexité $O()$, ce qui est relativement efficace. Cependant, à chaque changement de B, il faut reclaculer totalement les matrices U et e, ce qui est couteux en temps de calcul.

4.4 Factorisation "LU"

Introduction

Afin d'éviter de recalculer U si B change, on exprime A de notre expression initiale :

$$AX = B$$

En cette expression :

$$A = LU$$

Où L est une matrice triangulaire supérieure carrée et U une matrice inférieure carrée. Cette opération sera réalisé par la fonction "LU". On a donc :

$$LUX = B$$

On peut maintenant passer à la résolution qui va se faire en deux étapes. Tout d'abord, nous allons exprimer $Y = UX$ tel que :

$$LY = B$$

L étant une matrice triangulaire inférieure carrée, il sufit d'utiliser la fonction « Sol_Sup » afin d'obtenir Y.

Enfin, comme $UX = Y$ avec U une matrice supérieure carrée, il suffit d'utiliser « Sol_Inf » pour obtenir X.

Méthode

Code

```
void LU(double **L, double **A, int Taille)
{
    Nettoyer_Matrice(L, Taille, Taille);

    // Rempli les 1 en diagonale
    for (int i = 0; i < Taille; i++)
    {
        L[i][i] = 1;
    }

    // Calcule L et U
    for (int i = 0; i < Taille - 1; i++)
    {
        for (int k = i + 1; k < Taille; k++)
        {
            double C = A[k][i] / A[i][i];

            L[k][i] = C;

            for (int j = 0; j < Taille; j++)
            {
                A[k][j] = A[k][j] - (C * A[i][j]);
            }
        }
    }
}
```

Conclusion

Cette fonction est de complexité temporelle $O\,()$. Cependant, dans le cas où A est symétrique, il est possible de procéder à d'autres optimisations et diminuer le temps de calcul.

4.5 Factorisation de Cholesky

Méthode

Comme vu précédemment, nous avons réussi à optimiser notre résolution dans le cas où B change afin d'éviter de tout recalculer. Dans le cas où A est symétrique, il est également possible de procéder à d'autres optimisations :

On a donc la matrice A qui est :

— Symétrique : c'est à dire que $A = A^T$.

— Définie positive, c'est à dire quelle est positive et inversible, tel que $\langle AY, Y\rangle > 0, \forall Y \in \mathbb{R}^N \setminus \left\{\overrightarrow{0}\right\}$

On peut alors exprimer A en fonction de L une matrice triangulaire inférieure carrée dont la diagonale est positive :

$$A = L \cdot L^T$$

Si on remplace dans notre expression initiale, on a :

$$AX = B \Leftrightarrow L \cdot L^T \cdot X = B$$

On peut passer maintenant passer à la résolution. De manière similaire à la méthode « LU », nous allons procéder en deux étapes.

— Tout d'abord, la résolution de l'expression $LY = B$ où $Y = L^T X$ avec la fonction « Sol_Inf ». L étant une matrice triangulaire inférieure carrée.

— Enfin, la résolution de l'expression $L^T X = Y$ avec la fonction « Sol_Sup ». L^T étant également une matrice triangulaire inférieure carrée.

Application

Soit $N = 4$, L une matrice de taille $N * N$:

On créer une matrice L tel que $L \cdot L^T = A$, on a :

$$\underbrace{\begin{pmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} l_{11} & l_{21} & l_{31} & l_{41} \\ 0 & l_{22} & l_{32} & l_{42} \\ 0 & 0 & l_{33} & l_{43} \\ 0 & 0 & 0 & l_{44} \end{pmatrix}}_{L^T} = \underbrace{\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}}_{A}$$

$$\Leftrightarrow \begin{pmatrix} l_{11}^2 & l_{11}l_{21} & l_{11}l_{31} & l_{11}l_{41} \\ l_{21}l_{11} & l_{21}^2 + l_{22}^2 & l_{21}l_{31} + l_{22}l_{32} & l_{21}l_{41} + l_{22}l_{42} \\ l_{31}l_{11} & l_{31}l_{21} + l_{32}l_{22} & l_{31}^2 + l_{32}^2 + l_{33}^2 & l_{31}l_{41} + l_{32}l_{42} + l_{33}l_{43} \\ l_{41}l_{11} & l_{41}l_{21} + l_{42}l_{22} & l_{41}l_{31} + l_{42}l_{32} + l_{43}l_{33} & l_{41}^2 + l_{42}^2 + l_{43}^2 + l_{44}^2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

On a donc pour les termes en diagonale :

— $l_{11}^2 = a_{11} \Rightarrow l_{11} = \sqrt{a_{11}}$
— $l_{21}^2 + l_{22}^2 = a_{22} \Rightarrow l_{22} = \sqrt{a_{22} - l_{21}^2}$
— $l_{31}^2 + l_{32}^2 + l_{33}^2 = a_{33} \Rightarrow l_{33} = \sqrt{a_{33} - l_{31}^2 - l_{32}^2}$
— $l_{41}^2 + l_{42}^2 + l_{43}^2 + l_{44}^2 = a_{44} \Rightarrow l_{44} = \sqrt{a_{44} - l_{41}^2 - l_{42}^2 - l_{43}^2}$

On constate que : $l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}$

On a également pour les autres termes :
— $l_{21}l_{11} = a_{21} \Rightarrow l_{21} = \frac{a_{21}}{l_{11}}$
— $l_{31}l_{11} = a_{31} \Rightarrow l_{31} = \frac{a_{31}}{l_{11}}$
— $l_{31}l_{11} = a_{41} \Rightarrow l_{41} = \frac{a_{31}}{l_{11}}$

On constate que $l_{ij} = \frac{a_{ij}}{l_{jj}} - \sum_{k=1}^{j-1} l_{ik}l_{jk}$

Algorithme

`Ecrire code`

Conclusion

5 Méthodes itératives

Contrairement aux méthodes directes précédents où l'on cherche directement le vecteur X pour résoudre $AX = B$. Ici, nous allons nous intéresser.

5.1 Méthode de Jacobi

Méthode

Soit A une matrice carrée de taille N et B un vecteur de taille N, on cherche toujours à résoudre l'équation $AX = B$.

On peut décomposer A en 3 matrices tel que $A = E + F + G$:

— D la matrice contenant la diagonnale de A : $D = \begin{pmatrix} A_{11} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & A_{nn} \end{pmatrix}$.

— E la matrice contenant la partie inférieur de A (sans sa diagonale) : $E = \begin{pmatrix} 0 & \cdots & \cdots & 0 \\ A_{1j} & \ddots & \iddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ A_{ij} & \cdots & A_{(i-1)j} & 0 \end{pmatrix}$

— F la matrice contenant la partie supérieure de A (sans la diagonale) : $F = \begin{pmatrix} 0 & A & \cdots & A_{ij} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \iddots & \ddots & A \\ 0 & \cdots & \cdots & 0 \end{pmatrix}$

L'équation initiale $AX = B$ devient alors : $(E + D + F)\, X = B$, ce qui donne

$$\Rightarrow DX = b - (E + F)\, X$$

$$\Rightarrow X = D^{-1}\left(b - [E + F]\, X\right)$$

$$\begin{cases} X^0 & k = 0 \\ X^{k+1} = D^{-1}\left(b - [E + F]\, X^k\right) & k = 1, \ldots, N \end{cases}$$

Ainsi, pour un X^0 donné, on peut calculer $X_i^{k+1} = \frac{b_i - \sum_{j=1, j\neq i}^{N} A_{ij} X_j^k}{A_{ii}}$ pour $i = 1, \ldots, N$.

On peut alors calculer la norme : $\text{Norm} = \|x_{k+1} - x_k\|_2 = \sqrt{\sum_{n=0}^{N-1} \left(x_{k+1}[n] - x_k[n]\right)^2}$

On définit $\varepsilon = 0,001$ et $X_k = X_0$.

Algorithme

5.2 Méthode de Gauss-Seidel

Méthode

d

Algorithme

f

6 Code complet

7 Conclusion

Ainsi, nous avons trouvé différentes méthodes pour résoudre l'équation de matrice : $AX = B$ sans à calculer directement A^{-1}.