Rapport de projet mathématiques : Méthodes numériques pour la résolution de systèmes linéaires

Alix ANNERAUD - Amandine BURCON - Myriem ABID

Table des matières

1 Introduction

1.1 Motivation

La résolution des systèmes linéaires est utile à un bon nombre de domaines. Voici quelques exemples de problèmes que nous avons rencontrés précédemment :

- Électricité (P3) : Calcul d'intensité dans un circuit électronique en utilisant les lois des nœuds / mailles.
- Chimie (C3) : Calcul de l'équilibre des équations des réactions chimiques.
- Algèbre linéaire (M4) : Recherche des vecteurs propres d'une matrice carrée associés à une valeur propre donnée, déterminer le noyau d'un endomorphisme en dimension finie (système homogène).
- Géométrie analytique (Projet Informatique) : Déterminer la position relative de droites et de plans dans l'espace.

Nous allons nous intéresser au dernier cas, où nous devions, en projet informatique, détecter et déterminer la collision entre un parallélépipède rectangle ($IJKLMNOP$) et une droite (D).

En images de synthèse, les objets dans l'espace sont définis par des points dans $\mathbb{R}^3$.

Ainsi, pour une droite définie par les points A et B, il est relativement aisé de retrouver son vecteur directeur :

$$\overrightarrow{D} = \overrightarrow{AB} = \begin{pmatrix} x_B - x_A \\ y_B - y_A \\ z_B - z_A \end{pmatrix}$$

Ensuite, à partir du parallélépipède rectangle, on peut déterminer une base de l'espace :

$$\alpha \times \overrightarrow{JI} + \beta \times \overrightarrow{JK} + \gamma \times \overrightarrow{JL} = \alpha \times \begin{pmatrix} x_I - x_J \\ y_I - y_J \\ z_I - z_J \end{pmatrix} + \beta \times \begin{pmatrix} x_K - x_J \\ y_K - y_J \\ z_K - z_J \end{pmatrix} + \gamma \begin{pmatrix} x_L - x_J \\ y_L - y_J \\ z_L - z_J \end{pmatrix}, \forall \alpha, \beta, \gamma \in \mathbb{R}$$

Ainsi, on peut exprimer le vecteur $\overrightarrow{D}$ dans cette base, c'est à dire déterminer les valeurs de α, β et γ :

$$\alpha \times \overrightarrow{JI} + \beta \times \overrightarrow{JK} + \gamma \times \overrightarrow{JL} = \overrightarrow{D}$$

$$\alpha \times \begin{pmatrix} x_I - x_J \\ y_I - y_J \\ z_I - z_J \end{pmatrix} + \beta \times \begin{pmatrix} x_K - x_J \\ y_K - y_J \\ z_K - z_J \end{pmatrix} + \gamma \begin{pmatrix} x_L - x_J \\ y_L - y_J \\ z_L - z_J \end{pmatrix} = \begin{pmatrix} x_B - x_A \\ y_B - y_A \\ z_B - z_A \end{pmatrix}$$

On a donc une équation matricielle de la forme :

$$AX = B$$

$$\underbrace{\begin{pmatrix} x_I - x_J & x_K - x_J & x_L - x_J \\ y_I - y_J & y_K - y_J & y_L - y_J \\ z_I - z_J & z_K - z_J & z_L - z_J \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix}}_{X} = \underbrace{\begin{pmatrix} x_B - x_A \\ y_B - y_A \\ z_B - z_A \end{pmatrix}}_{B}$$

Une fois trouvés α, β et γ (X), on connaît alors la position relative de la droite D par rapport au parallélépipède rectangle. Ainsi, avec de simples comparaisons, on peut déduire si la droite est en collision avec le parallélépipède rectangle, et si oui, à quels endroits.

1.2 Théorie

Notre but sera de résoudre pour X l'équation linéaire de matrices $AX = B$ dans $\mathbb{R}^N$ sans avoir à calculer A^{-1}. En effet, trouver l'inverse de A peut poser problème de par le temps de calcul lorsque N devient très grand. Par exemple, pour une taille $N = 20$, il faudrait 5 fois l'âge de l'univers ($5 \times 13,7 \times 10^9$ années) pour calculer A^{-1}. Nous allons donc, à l'aide de plusieurs méthodes, tenter de résoudre l'équation sans réaliser ce calcul, en décomposant le problème en sous-problèmes plus simples à résoudre. Nous procéderons à des optimisations graduelles d'une méthode à l'autre du coût en temps de calcul et en espace mémoire utilisé pour le calcul. On utilisera tout au long de ce rapport les matrices $A \in \mathbb{R}^{N \times N}$ et $B, X \in \mathbb{R}^N$.

2 Biographies

Johann Carl Friedrich 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.

André-Louis 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 contributions 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.

Charles Gustave Jacob Jacobi

Charles Gustave Jacob Jacobi est un mathématicien allemand né le 10 décembre 1804. Après avoir soutenu une thèse sur la théorie des fractions, il enseigne les mathématiques à l'université de Königsberg. En parallèle de sa fonction de professeur, il effectue des travaux de recherche en physique mathématique, en théorie des nombres et en analyse mathématique pour n'en citer que quelques-uns. On lui doit notamment la théorie des déterminants et en particulier l'invention du déterminant d'une matrice (jacobienne). Il publie également un traité fondamental sur les fonctions elliptiques qui révolutionne la physique mathématique mais il dédie aussi un ouvrage aux équations différentielles. Malgré sa mort prématurée en 1851, à l'âge de 46 ans, ses travaux furent nombreux et on retrouve parmi eux une méthode de résolution de systèmes linéaires. Contrairement aux méthodes de Gauss, de Cholesky ou encore de factorisation LU qui sont des méthodes directes, la méthode de Jacobi est une méthode dite itérative. Il est alors pertinent de s'intéresser à cette méthode et de l'implémenter en C si l'on veut réaliser une comparaison entre les deux types de méthodes.

Philipp von Seidel

Philipp von Seidel est un mathématicien et physicien Allemand. Né le 24 octobre 1821, il étudie dans de nombreuses villes dû au travail de son père, puis suit des cours privés sous la tutelle de L.C. Schnürlein, ancien élève de Gauss, pendant un an. Il suit ensuite l'enseignement d'autres grands représentants de leur discipline tels que Dirichlet, Encke, Jacobi ou encore Neumann dans trois universités différentes. En 1846, Seidel obtient son doctorat grâce à sa thèse : "Sur la meilleure forme des miroirs dans un télescope", puis publie une autre thèse six mois plus tard sur un sujet entièrement différent : "Études sur la convergence et

divergence des fractions continues", ce qui lui permet de devenir professeur à l'université de Munich. Des problèmes de vue le forcent à prendre une retraite anticipée, et n'étant pas marié, ce sont sa sœur puis une veuve qui s'occupent de lui en fin de vie. Il meurt le 13 août 1896.

Seidel se concentre tout au long de sa carrière sur ses travaux en optique et en astronomie ainsi que sur l'analyse mathématique, et décompose les aberrations optiques du premier ordre en cinq équations, appelées "équations de Seidel". Il applique la théorie des probabilités à l'astronomie, et l'utilise également pour étudier la fréquence de certaines maladies ou le climat. Il est particulièrement connu pour la méthode de Gauss-Seidel de résolution d'équation numérique, que nous décrirons dans ce rapport.

3 Préliminaires

3.1 Type de données

Tout d'abord, en C comme dans 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 parmi 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 chiffres 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 » ne nécessite pas de temps processeur supplémentaire par rapport à un type « float » sur les ordinateurs modernes, pour lesquels 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 a 10 - 20 ans car les processeurs plus anciens possédaient seulement des unités et registres de calcul flottant de 32 bits, 16 bits voire 8 bits (pour les plus anciens). Ainsi, il fallait des cycles d'horloges supplémentaires pour un calcul flottant sur 64 bits par rapport à un calcul flottant sur 32 bits. Idem dans le cas de calculateurs scientifiques qui possèdent généralement des unités de calculs et registres flottants de 128 bits (voire plus). 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 Structure des données

Ensuite, les concepts de matrice et de vecteur ne sont pas directement implémentés dans le langage C. Cependant, nous pouvons utiliser une liste simple de « double » pour les vecteurs et des listes de « double » imbriquées 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);
    }
```

```
    return Matrice;
}
```

Ces fonctions appellent les fonctions “Nettoyage_Vecteur” et “Nettoyage_Matrice”, qui vont remplir de 0 les matrices et vecteurs. En effet, la mémoire alloué par le système n’est pas toujours « propre » ; elle peut 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 Nettoyage_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 :

```
void Afficher_Vecteur(double *Vecteur, int Taille)
{
    // 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++)
    {

        // 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, la libération de la mémoire (dés-allocation) doit être faite manuellement :

```
void Desallocation_Vecteur(double *Vecteur)
{
    free(Vecteur);
}

void Desallocation_Matrice(double **Matrice, int Taille)
{
    // Itère parmis la première dimension de la matrice.
    for (int i = 0; i < Taille; i++)
    {
        Desallocation_Vecteur(Matrice[i]);
    }
}
```

4 Méthodes directes

On considère qu'une méthode de résolution de système linéaire $AX = B$ est directe lorsque celle-ci permet d'aboutir à une solution, si cette dernière existe, au terme d'un nombre fini d'opérations élémentaires, c'est-à-dire les additions, soustractions, multiplications et divisions. Les solutions obtenues via ce type de méthodes sont exactes mais cela ne signifie pas qu'elles sont infaillibles. En effet, plus le nombre d'opérations à effectuer est important, plus le risque d'avoir des erreurs de calcul augmente. Généralement, ce nombre est proportionnel à la taille du système étudié. Il existe de nombreuses méthodes directes, nous en avons sélectionné quelques-unes que nous décrirons dans ce rapport. Nous commencerons par introduire deux algorithmes, un premier pour le cas des matrices triangulaires inférieures et un second pour les matrices triangulaires supérieures. Nous pourrons ensuite nous intéresser à la méthode d'élimination de Gauss, à la factorisation LU ainsi qu'à la méthode de Cholesky.

4.1 Méthode triangulaire inférieure

Principe

Pour cette première méthode, A est une matrice triangulaire inférieure. On appelle matrice triangulaire inférieure toute matrice carrée à coefficients dans $\mathbb{R}$ dont la partie située au-dessus de la diagonale principale est uniquement constituée de coefficients nuls : $\forall i < j, a_{ij} = 0$. On peut donc facilement trouver X en suivant un algorithme dit « de la descente ».

Méthode

Soit $N = 3$, on a :

$$AX = B$$

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

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

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

On en déduit alors une formule générale pour les coefficients de 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 Somme = 0;
        for (int j = 0; j < i; j++)
        {
            Somme = Somme + A[i][j] * X[j];
        }
        // Calcul du terme X[i].
        X[i] = (B[i] - Somme) / A[i][i];
    }

    return X;
}
```

Exemple

Pour $A = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 1 & 4 & -1 \end{pmatrix}$ et $B = \begin{pmatrix} 1 \\ 8 \\ 10 \end{pmatrix}$, l'algorithme nous retourne $X = \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix}$, ce qui est correct.

Conclusion

Ainsi, cet algorithme possède une complexité temporelle maximale quadratique $O\left(N^2\right)$ car nous avons deux boucles imbriquées l'une dans l'autre qui dépendent de la taille N de A,B et X. Nous nous servirons de cet algorithme dans les méthodes qui suivent dont le but sera de se rapporter à une forme de matrice triangulaire.

4.2 Méthode triangulaire supérieure

Principe

Cette méthode est similaire à la précédente, mais A est ici une matrice triangulaire supérieure. On appelle matrice triangulaire supérieure toute matrice carrée à coefficients dans $\mathbb{R}$ dont la partie située en-dessous de la diagonale principale est uniquement constituée de coefficients nuls : $\forall i > j, a_{ij} = 0$. On va donc cette fois trouver X en suivant un algorithme dit « de la remontée ».

Méthode

Soit $N = 3$, on a :

$$AX = B$$

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

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

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

On en déduit alors une formule générale pour les coefficients de X :

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

Code

```
double *Sol_Sup(double **A, double *B, int Taille)
{
    // Allocation du vecteur X.
    double *X = Allocation_Vecteur(Taille);

    // Calcul du dernier terme de X (terme initial).
    X[Taille - 1] = B[Taille - 1] / A[Taille - 1][Taille - 1];

    // Itère parmis les lignes de la matrice.
    for (int i = Taille - 2; i >= 0; i--)
    {
        // Calcul de la somme des a[i][j] * x[j]
        double Somme = 0;
        for (int j = i + 1; j < Taille; j++)
        {
            Somme += A[i][j] * X[j];
        }

        X[i] = (B[i] - Somme) / A[i][i];
    }

    return X;
}
```

Exemple

Ainsi, pour $A = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 8 \\ 0 & 0 & 5 \end{pmatrix}$ et $B = \begin{pmatrix} 6 \\ 16 \\ 15 \end{pmatrix}$, l'algorithme nous retourne $X = \begin{pmatrix} 1 \\ -2 \\ 3 \end{pmatrix}$, ce qui est correct.

Conclusion

De la même manière que le précédent, cet algorithme possède une complexité quadratique ($O\left(N^2\right)$).

4.3 Élimination de Gauss

Principe

On veut maintenant résoudre l'équation $AX = B$ lorsque A est une matrice carrée quelconque. Nous allons utiliser la méthode de l'élimination de Gauss pour transformer l'expression $AX = B$ en $UX = e$ avec $U \in \mathbb{R}^{N \times N}$, une matrice triangulaire supérieure. Ainsi, il ne restera plus qu'à résoudre $UX = e$ avec la fonction « Sol_Sup » pour obtenir X.

Méthode

Soit $N = 3$, on a :

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

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) :

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

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

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

On remarque que pour éliminer un x_i d'une équation (2) en utilisant (1) (équation précédente), on utilise une formule très simple : $(2) - \frac{a_1}{a_2} \times (1)$ avec a_k le coefficient de x_i dans l'équation (k).

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

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

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

Code

```
void Gauss(double **A, double *B, double **U, double *e, int Taille)
{
    // Itère parmis la première dimension de la matrice.
    for (int i = 0; i < Taille - 1; i++)
    {
        // Itère parmis la deuxième dimension de la matrice.
        for (int k = i + 1; k < Taille; k++)
        {
            // Calcul du coefficient C = A_k_i / A_i_i.
            double C = A[k][i] / A[i][i];
            // Remplacement des valeurs de A.
            for (int j = 0; j < Taille; j++)
            {
                U[k][j] = A[k][j] - (C * A[i][j]);
            }
            // Remplacement des valeurs de B.
            e[k] = B[k] - (C * B[i]);
        }
    }
}
```

Exemple

Ainsi, pour $A = \begin{pmatrix} 1 & 2 & 3 \\ 5 & 2 & 1 \\ 3 & -1 & 1 \end{pmatrix}$ et $B = \begin{pmatrix} 5 \\ 5 \\ 6 \end{pmatrix}$, l'algorithme nous retourne $X = \begin{pmatrix} 1 \\ -2 \\ 2 \end{pmatrix}$, ce qui est correct.

Conclusion

Cette méthode est de complexité $O\left(\frac{2N^3}{3}\right)$, ce qui est relativement efficace. Cependant, elle demande de modifier entièrement A et B. Lors de résolution de problèmes, il arrive que certaines données changent, ce qui modifierait A mais plus souvent B. Il faudrait donc recalculer entièrement l'équation $UX = e$ puis la solution X à chaque modification, ce qui est coûteux en temps de calcul.

4.4 Factorisation "LU"

Principe

Afin d'éviter de recalculer U et e lorsque B change, on factorise A en LU, où $L \in \mathbb{R}^{N \times N}$, une matrice triangulaire inférieure et $U \in \mathbb{R}^{N \times N}$, une matrice triangulaire supérieure. Cette opération sera réalisée 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 poser $Y = UX$. Ce qui donne : $LY = B$. L étant une matrice triangulaire inférieure, il suffit d'utiliser la fonction « Sol_Inf » afin d'obtenir Y.
- Enfin, comme $UX = Y$ avec U une matrice supérieure, on peut utiliser « Sol_Sup » pour obtenir X.

Méthode

Pour faciliter les calculs, on décide que la diagonale de L sera composée de 1. On utilise ensuite l'élimination de Gauss pour calculer L, mais sans modifier la matrice B. Cela revient à remplacer les coefficients de A, pour créer U, par les combinaisons linéaires de l'élimination de Gauss, selon la formule $a_{kj} = a_{kj} - \left(\frac{a_{ki}}{a_{ii}} \times a_{ij}\right)$. On complète en même temps L par les coefficients de ces combinaisons linéaires : $l_{ki} = \frac{a_{ki}}{a_{ii}}$.

Code

```
void LU(double **L, double **A, int Taille)
{
    Nettoyage_Matrice(L, 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]);
                }
            }
        }
}
```

Exemple

— Pour $A = \begin{pmatrix} 3 & 1 & 2 \\ 3 & 2 & 6 \\ 6 & 1 & -1 \end{pmatrix}$, l'algorithme nous retourne $L = \begin{pmatrix} 1 & 0 & 0 \\ 5 & 1 & 0 \\ 3 & 0,875 & 1 \end{pmatrix}$ et $U = \begin{pmatrix} 1 & 2 & 3 \\ 0 & -8 & -14 \\ 0 & 0 & 4,25 \end{pmatrix}$.

— Puis, pour $B = \begin{pmatrix} 6 \\ 16 \\ 15 \end{pmatrix}$ et L, « Sol_Sup » retourne $Y = \begin{pmatrix} 5 \\ -20 \\ 8,5 \end{pmatrix}$.

— Enfin, pour U et Y, « Sol_Inf » retourne $X = \begin{pmatrix} 1 \\ -1 \\ 2 \end{pmatrix}$, ce qui est correct.

Conclusion

Cette fonction est de complexité temporelle $O\left(\frac{N^3}{3}\right)$. 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

Principe

Cette nouvelle méthode nous permet d'optimiser encore plus le code dans le cas où A est symétrique ($A = A^T$) et définie positive (positive et inversible $\Leftrightarrow \langle AY, Y\rangle > 0, \forall Y \in \mathbb{R}^N \setminus \left\{\overrightarrow{0}\right\}$). On peut alors exprimer A en fonction de $L \in \mathbb{R}^{N\times N}$ une matrice triangulaire inférieure dont la diagonale est strictement positive : $A = L \cdot L^T$. La transposée de L étant une matrice L^T où les lignes et les colonnes de L ont été permutées. Si on remplace dans notre expression initiale, on a : $L \cdot L^T \cdot X = B$. On peut ensuite passer à la résolution. De manière similaire à la méthode « LU », nous allons procéder en deux étapes.

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

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

Méthode

Soit $N = 4$, on cherche une matrice L tel que $L \cdot L^T = A$, on a :

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

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

— On a donc pour les termes en diagonale : $l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}$

— On a également pour les autres termes : $l_{ij} = \frac{a_{ij}}{l_{jj}} - \sum_{k=1}^{j-1} l_{ik} l_{jk}$

Code

```
double **Cholesky(double **A, int Taille)
{
    // Allocation de la matrice L.
    double **L = Allocation_Matrice(Taille);

    // Itère parmis les lignes de la matrice.
    for (int j = 0; j < Taille; j++)
    {
        // Calcul de la somme.
        double Somme = 0;
        for (int k = 0; k < j; k++)
        {
            Somme += L[j][k] * L[j][k];
        }

        // Calcul des termes de la diagonale de L.
        L[j][j] = sqrt(A[j][j] - Somme);

        // Calcul des termes pour i = j+1 à N.
        for (int i = j + 1; i < Taille; i++)
        {
            Somme = 0;
            for (int k = 0; k < j; k++)
            {

                Somme += L[i][k] * L[j][k];
            }
            L[i][j] = (A[i][j] - Somme) / L[j][j];
        }
    }
    return L;
}

double **Transposer(double **Matrice, int Taille)
{
    // On itère parmis les colones.
    for (int i = 0; i < Taille; i++)
    {
        // On itère parmis les lignes.
        for (int j = i + 1; j < Taille; j++)
        {
            // On inverse les termes (en cooordonnées) de la matrice.
            Matrice[i][j] = Matrice[j][i];
            //  On supprime les termes inférieurs.
            Matrice[j][i] = 0;
        }
    }
    return Matrice;
```

```
}
```

Exemple

— Ainsi, pour $A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 5 & 5 & 5 \\ 1 & 5 & 14 & 14 \\ 1 & 5 & 14 & 15 \end{pmatrix}$, l'algorithme nous retourne $L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 \\ 1 & 2 & 3 & 0 \\ 1 & 2 & 3 & 1 \end{pmatrix}$.

— Puis, pour $B = \begin{pmatrix} 5 \\ 1 \\ 3 \\ 1 \end{pmatrix}$ et L, « Sol_Inf » retourne $Y = \begin{pmatrix} 5 \\ -20 \\ 8,5 \end{pmatrix}$.

— Enfin, pour L^T et Y, « Sol_Sup » retourne $X = \begin{pmatrix} 1 \\ -1 \\ 2 \end{pmatrix}$, ce qui est correct.

Conclusion

Cette méthode qui est un cas particulier de la précédente est la moins complexe avec $O\left(\frac{N^3}{6}\right)$, ce qui en fait la plus pertinente pour résoudre l'équation. Cependant elle n'est utilisable que lorsque A est symétrique et définie positive, son utilisation est donc restreinte.

4.6 Conclusion

Dans cette partie, nous avons pu voir plusieurs manières de résoudre un système linéaire par le biais d'opérations élémentaires. Même si nous avons vu que certaines méthodes étaient moins coûteuses que d'autres en matière de temps de calcul, le choix de la méthode dépend également des spécificités du système linéaire. Par exemple, dans le cas d'une matrice définie positive mais qui n'est pas symétrique, la méthode de Cholesky ne pourra pas être appliquée et il faudra passer par une autre méthode. De plus, leurs résultats sont justes lorsque la taille de la matrice et des vecteurs reste raisonnable car les risques d'erreurs de calcul sont proportionnels à la taille. Ces méthodes ne sont pas les plus économes en matière d'utilisation d'espace mémoire, c'est pourquoi nous allons étudier un autre type de méthodes de résolution de systèmes linéaires.

5 Méthodes itératives

Les méthodes itératives sont une autre approche à la résolution de systèmes linéaires $AX = B$. Afin d'aboutir à une solution via ces méthodes, on part d'une approximation de solution que l'on considérera comme une ébauche et non comme notre résultat final, on peut par exemple l'obtenir à l'aide d'une méthode directe . On effectue ensuite des itérations qui établissent une suite de solutions intermédiaires qui se rapprochent de plus en plus de la solution finale. L'idée est donc de construire une suite de vecteurs intermédiaires $\left(X^k\right)$ qui converge vers X, solution du système. Procéder de cette manière permet de limiter la propagation d'erreurs et ainsi obtenir la solution la plus proche possible de la solution réelle. Nous avons sélectionné deux méthodes itératives à décrire, la méthode de Jacobi et la méthode de Gauss-Seidel. Même si nous n'avons pas eu l'occasion d'aborder en détails la méthode de Gauss-Seidel en cours, il nous semblait pertinent de tout de même l'inclure dans notre rapport afin d'avoir un élément supplémentaire de comparaison dans notre étude des différentes méthodes de résolution de systèmes linéaires.

5.1 Méthode de Jacobi

Principe

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

— $D \in \mathbb{R}^{N\times N}$, la matrice contenant la diagonale de A : $D = \begin{pmatrix} A_{11} & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & A_{nn} \end{pmatrix}$. D est inversible.

— $E \in \mathbb{R}^{N\times N}$, la matrice contenant la partie inférieure de A (sans la diagonale) : $E = \begin{pmatrix} 0 & \cdots & \cdots & 0 \\ A_{1j} & \ddots & \iddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ A_{ij} & \cdots & A_{(i-1)j} & 0 \end{pmatrix}$.

— $F \in \mathbb{R}^{N\times N}$, 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 :

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

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

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

Si il y a convergence de la suite (X^n), alors $X^{k+1} \approx X^k = X$.

Méthode

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 définit alors un ε pour que la boucle s'arrête une fois que le résultat est considéré comme satisfaisant, c'est à dire : $\left\|X^{k+1} - X^k\right\|_2 = \sqrt{\sum_{n=0}^{N-1} \left(X^{k+1}[n] - X^k[n]\right)^2} \leqslant \varepsilon$. On définit également un nombre d'itérations maximal dans le cas où l'algorithme n'atteint pas ε.

Code

```
double *Jacobi(double **A, double *B, int Taille)
{
    // Allocation du vecteur X_k.
    double *X_k = Allocation_Vecteur(Taille);

    // Remplissage de X_k par des 1 (X_0).
    for (int i = 0; i < Taille; i++)
    {
        X_k[i] = 1;
    }

    // Allocation du vecteur X_k_1.
    double *X_k_1 = Allocation_Vecteur(Taille);

    double Norme;
    int it = 0;
    int it_max = 10;
    double Epsilon = 0.001;

    // Itération.
    do
    {
        // Calcul de X_k_1.
        for (int i = 0; i < Taille; i++)
        {
            // Calcul de la somme des a[i][j] * x_k[j].
            double Somme = 0;
            for (int j = 0; j < Taille; j++)
            {
                if (i != j)
                {
                    Somme += A[i][j] * X_k[j];
                }
            }
            X_k_1[i] = (B[i] - Somme) / A[i][i];
        }

        // Calcul de la norme.
        Norme = 0;
        for (int i = 0; i < Taille; i++)
        {
            Norme += (X_k_1[i] - X_k[i]) * (X_k_1[i] - X_k[i]);
        }
        Norme = sqrt(Norme);

        // On remplace X_k par X_k_1.
        for (int i = 0; i < Taille; i++)
        {
            X_k[i] = X_k_1[i];
        }
        it++;
    } while ((Norme > Epsilon) && (it < it_max));
```

```
    return X_k_1;
}
```

Exemple

Pour $A = \begin{pmatrix} 4 & 1 & 1 \\ 1 & 3 & 1 \\ 2 & 0 & 5 \end{pmatrix}$, $B = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$, $X^0 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$ $\varepsilon = 0,001$ et un nombre d'itérations maximal de 10, l'algorithme retourne $X = \begin{pmatrix} 0,157701 \\ 0,236300 \\ 0,137924 \end{pmatrix}$, ce que l'on peut considérer comme correct car : $X \cdot A = \begin{pmatrix} 1,005028 \\ 1,004525 \\ 1,005022 \end{pmatrix} \approx B = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$.

Conclusion

Cet algorithme a une complexité $O\left(3N^2 + 2N\right)$, ce qui est moindre que pour les méthodes précédentes. On peut également décider, dans une certaine mesure, de la précision voulue du résultat en modifiant ε. Cependant X^0 est choisi arbitrairement, ce qui réduit la précision des résultats. Le nombre d'itérations nécessaires à un résultat précis peut également être un point négatif.

5.2 Méthode de Gauss-Seidel

Principe

Dans la méthode précédente, les coefficients du vecteur X^{k+1} sont plus précis que ceux de X^k. Lorsque l'on calcule X^{k+1}, on calcule les coefficients de ce vecteur un par un : on peut donc utiliser les coefficients de ce vecteur déjà calculés plutôt que ceux du vecteur précédent. Une fois que la formule générale est trouvée, on suit les mêmes étapes de comparaison de la norme avec un epsilon choisi, et on a de la même façon X qui sera la limite d'une suite avec X^0 choisi.

Méthode

En calculant X_i^{k+1} on ne connaît les coefficients de X^{k+1} que jusqu'à $i-1$, on utilise ceux de X^k pour le reste de la somme. On sépare donc la somme de la formule précédente en deux. On a : $X_i^{k+1} = \frac{b^i - \sum_{j=1}^{i-1} a_{ij} X_j^{k+1} - \sum_{j=i+1}^{N} a_{ij} X_j^k}{a_{ii}}$, pour $i = 1, \ldots, N$.

Code

```
double *Gauss_Seidel(double **A, double *B, int Taille)
{
    // Allocation des vecteurs.
    double *X_k = Allocation_Vecteur(Taille);
    double *X_k_1 = Allocation_Vecteur(Taille);

    // Remplissage de X_k par des 1 (X_0).
    for (int i = 0; i < Taille; i++)
    {
        X_k[i] = 1;
    }

    double Norme;
```

```
    int it = 0;
    int it_max = 10;
    double Epsilon = 0.001;

    do
    {
        // Calcul de X_k_1.
        for (int i = 0; i < Taille; i++)
        {
            // Calcul de la somme des A[i][j] * X_k_1[j].
            double Somme = 0;
            for (int j = 0; j < i; j++)
            {
                Somme += A[i][j] * X_k_1[j];
            }
            // Calcul de la somme des a[i][j] * x_k[j].
            for (int j = i + 1; j < Taille; j++)
            {
                Somme += A[i][j] * X_k[j];
            }
            X_k_1[i] = (B[i] - Somme) / A[i][i];
        }

        // Calcul de la norme entre X_k_1 et X_k.
        Norme = 0;
        for (int i = 0; i < Taille; i++)
        {
            Norme += (X_k_1[i] - X_k[i]) * (X_k_1[i] - X_k[i]);
        }
        Norme = sqrt(Norme);

        // On remplace X_k par X_k_1.
        for (int i = 0; i < Taille; i++)
        {
            X_k[i] = X_k_1[i];
        }
        it++;
    } while ((Norme > Epsilon) && (it < it_max));

    return X_k_1;
}
```

Exemple

Pour $A = \begin{pmatrix} 10 & 1 & 1 \\ 2 & 10 & 1 \\ 2 & 2 & 10 \end{pmatrix}$, $B = \begin{pmatrix} 12 \\ 13 \\ 14 \end{pmatrix}$, $X^0 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$, $\varepsilon = 0,001$ et un nombre d'itérations maximal de 10, l'algorithme retourne $X = \begin{pmatrix} 1,000000 \\ 1,000000 \\ 1,000000 \end{pmatrix}$, ce qui est correct car $X \cdot A = \begin{pmatrix} 12 \\ 13 \\ 14 \end{pmatrix} = B$.

Conclusion

La complexité de $O\left(3N^2 + 2N\right)$ est la même que précédemment, mais on gagne en espace mémoire puisque seulement un vecteur X est nécessaire, au lieu de deux auparavant. De plus, les résultats produits sont plus précis que la méthode de Jacobi.

5.3 Conclusion

Ici, nous avons étudié deux méthodes itératives visant à résoudre des systèmes linéaires. Comparées aux méthodes directes, celles-ci ne nécessitent pas de stocker les matrices entières en mémoire vive mais uniquement les termes non nuls, en d'autres termes, elles occupent moins d'espace mémoire. De plus, pour des systèmes très grands, elles permettent d'obtenir des solutions en passant par moins de calculs qu'avec les méthodes directes. En revanche, le résultat obtenu à la fin ne sera pas exact mais très proche de la solution réelle avec tout de même une propagation d'erreurs assez limitée. Les méthodes itératives sont donc moins coûteuses que les méthodes directes. Néanmoins, les conditions de convergence pour les suites établies ne sont pas faciles à mettre en œuvre.

6 Conclusion

Ainsi, nous avons étudié plusieurs méthodes possibles pour la résolution d'une équation de matrice de la forme $AX = B$, sans calculer A^{-1}. Voici un tableau récapitulatif des différentes méthodes :

Méthode	Complexité temporelle	Avantages	Inconvénients
Sol_Inf	$O\left(N^2\right)$		— Fonctionne si A est une matrice triangulaire inférieure.
Sol_Sup	$O\left(N^2\right)$		— Fonctionne si A est une matrice triangulaire supérieure.
Gauss	$O\left(\frac{2N^3}{3}\right)$	— Résolution d'une matrice quelconque (pivot non nul).	— Modifie totalement A et B.
LU	$O\left(\frac{N^3}{3}\right)$	— Optimisée même si B est modifié et si A reste constant.	— Nécessite la résolution de deux systèmes linéaires avec des matrices triangulaires. — Empreinte mémoire élevée : stockage des deux matrices L et U.
Cholesky	$O\left(\frac{N^3}{6}\right)$	— Optimisée même si B est modifié et si A reste constante et est symétrique.	— Ne permet pas la résolution pour A une matrice carrée quelconque. A doit être symétrique et définie positive. — Nécessite la résolution de deux systèmes linéaires avec des matrices triangulaires. — Empreinte mémoire élevée : stockage des deux matrices L et L^T.
Jacobi	$O\left(3N^2 + 2N\right)$	— Précision ajustable.	— Valeurs approchées.
Gauss-Seidel	$O\left(3N^2 + 2N\right)$	— Précision ajustable. — Empreinte mémoire réduite : un seul vecteur X. — Précision plus élevée que Jacobi.	— Valeurs approchées.

On constate que les méthodes ci-dessus sont complémentaires car elles présentent chacune des avantages et des inconvénients. Le choix de la méthode doit alors se faire en fonction de :

— La forme de A et B, afin d'optimiser le temps de calcul et éventuellement l'empreinte mémoire.

— La précision requise.

Il sera possible de résoudre tout système linéaire qui admet une solution dans $\mathbb{R}^N$ avec un de ces algorithmes, ainsi que par d'autres qui n'ont pas été citées dans ce rapport (factorisation QR, factorisation de Householder, relaxation).

Pour conclure, il en revient à l'utilisateur de choisir quelle méthode convient le mieux à sa situation, le but étant d'économiser le temps de calcul et le coût mémoire de notre résolution. Nous aurions pu éventuellement rédiger un algorithme qui fait ce choix automatiquement.

7 Code complet

```c
// - Importation des librairies nécéssaires.

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>

// - Gestion des vecteurs et matrices.

// - - Nettoyage

// Fonction qui nettoie un vecteur (remplissage par des 0).
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;
    }
}

// - Fonction qui nettoie une matrice (remplissage par des 0).
void Nettoyage_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);
    }
}

// - - Allocation

// Fonction qui alloue un vecteur de taille N.
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;
}

// Fonction qui alloue une matrice carrée de taille N * M
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);
    }

    return Matrice;
}

// - - Désallocation

// Fonction qui désalloue un vecteur de taille N.
void Desallocation_Vecteur(double *Vecteur)
{
    free(Vecteur);
}

// Fonction qui désalloue une matrice carrée de taille N.
void Desallocation_Matrice(double **Matrice, int Taille)
{
    // Itère parmis la première dimension de la matrice.
    for (int i = 0; i < Taille; i++)
    {
        Desallocation_Vecteur(Matrice[i]);
    }
}

// - - Affichage

// Fonction qui affiche un vecteur de taille N.
void Afficher_Vecteur(double *Vecteur, int Taille)
{
    // 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]);
    }
}

// Fonction qui affiche une matrice carrée de taille N.
void Afficher_Matrice(double **Matrice, int Taille)
{
    // Itère parmis la première dimension de la matrice.
    for (int i = 0; i < Taille; i++)
    {

        // 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");
    }
}

// - Méthodes directes de résolution.

// - - Resolution de matrices triangulaires carrées

// Fonction qui résoud AX = B avec A une matrice triangulaire inférieure
    carrée et B un vecteur (algorithme de la redescente).
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 Somme = 0;
        for (int j = 0; j < i; j++)
        {
            Somme = Somme + A[i][j] * X[j];
        }
        // Calcul du terme X[i].
        X[i] = (B[i] - Somme) / A[i][i];
    }

    return X;
}

// Fonction qui résoud AX = B avec A une matrice triangulaire supérieure
    carrée et B un vecteur (algorithme de la remontée).
double *Sol_Sup(double **A, double *B, int Taille)
{
    // Allocation du vecteur X.
    double *X = Allocation_Vecteur(Taille);

    // Calcul du dernier terme de X (terme initial).
    X[Taille - 1] = B[Taille - 1] / A[Taille - 1][Taille - 1];

    // Itère parmis les lignes de la matrice.
    for (int i = Taille - 2; i >= 0; i--)
    {
        // Calcul de la somme des a[i][j] * x[j]
        double Somme = 0;
        for (int j = i + 1; j < Taille; j++)
        {
            Somme += A[i][j] * X[j];
```

```
        }

        X[i] = (B[i] - Somme) / A[i][i];
    }

    return X;
}

// - - Elimination de Gauss

// Fonction qui effectue l'élimination de Gauss pour transformer A en U,
   une matrice triangulaire supérieure carrée (algorithme de Gauss).
void Gauss(double **A, double *B, double **U, double *e, int Taille)
{
    // Itère parmis la première dimension de la matrice.
    for (int i = 0; i < Taille - 1; i++)
    {
        // Itère parmis la deuxième dimension de la matrice.
        for (int k = i + 1; k < Taille; k++)
        {
            // Calcul du coefficient C = A_k_i / A_i_i.
            double C = A[k][i] / A[i][i];
            // Remplacement des valeurs de A.
            for (int j = 0; j < Taille; j++)
            {
                U[k][j] = A[k][j] - (C * A[i][j]);
            }
            // Remplacement des valeurs de B.
            e[k] = B[k] - (C * B[i]);
        }
    }
}

// - - Factorisation LU

// Fonction qui effectue la factorisation LU pour transformer A en L et U,
    deux matrices triangulaires carrées inférieures et supérieures (
   algorithme de LU).
void LU(double **L, double **A, int Taille)
{
    Nettoyage_Matrice(L, 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]);
            }
        }
    }
}

// - - Factorisation de Cholesky

// Fonction qui transforme A en L*L^T. Renvoi L.
double **Cholesky(double **A, int Taille)
{
    // Allocation de la matrice L.
    double **L = Allocation_Matrice(Taille);

    // Itère parmis les lignes de la matrice.
    for (int j = 0; j < Taille; j++)
    {
        // Calcul de la somme.
        double Somme = 0;
        for (int k = 0; k < j; k++)
        {
            Somme += L[j][k] * L[j][k];
        }

        // Calcul des termes de la diagonale de L.
        L[j][j] = sqrt(A[j][j] - Somme);

        // Calcul des termes pour i = j+1 à N.
        for (int i = j + 1; i < Taille; i++)
        {
            Somme = 0;
            for (int k = 0; k < j; k++)
            {

                Somme += L[i][k] * L[j][k];
            }
            L[i][j] = (A[i][j] - Somme) / L[j][j];
        }
    }
    return L;
}

// Fonction qui transpose une matrice carrée inférieure en matrice carrée
   supérieure.
double **Transposer(double **Matrice, int Taille)
{
    // On itère parmis les colones.
    for (int i = 0; i < Taille; i++)
    {
```

```
            // On itère parmis les lignes.
            for (int j = i + 1; j < Taille; j++)
            {
                // On inverse les termes (en cooordonnées) de la matrice.
                Matrice[i][j] = Matrice[j][i];
                //  On supprime les termes inférieurs.
                Matrice[j][i] = 0;
            }
        }
        return Matrice;
}

// - Méthodes itératives de résolution.

// - - Jacobi

double *Jacobi(double **A, double *B, int Taille)
{
    // Allocation du vecteur X_k.
    double *X_k = Allocation_Vecteur(Taille);

    // Remplissage de X_k par des 1 (X_0).
    for (int i = 0; i < Taille; i++)
    {
        X_k[i] = 1;
    }

    // Allocation du vecteur X_k_1.
    double *X_k_1 = Allocation_Vecteur(Taille);

    double Norme;
    int it = 0;
    int it_max = 10;
    double Epsilon = 0.001;

    // Itération.
    do
    {
        // Calcul de X_k_1.
        for (int i = 0; i < Taille; i++)
        {
            // Calcul de la somme des a[i][j] * x_k[j].
            double Somme = 0;
            for (int j = 0; j < Taille; j++)
            {
                if (i != j)
                {
                    Somme += A[i][j] * X_k[j];
                }
            }
            X_k_1[i] = (B[i] - Somme) / A[i][i];
        }

        // Calcul de la norme.
```

```
        Norme = 0;
        for (int i = 0; i < Taille; i++)
        {
            Norme += (X_k_1[i] - X_k[i]) * (X_k_1[i] - X_k[i]);
        }
        Norme = sqrt(Norme);

        // On remplace X_k par X_k_1.
        for (int i = 0; i < Taille; i++)
        {
            X_k[i] = X_k_1[i];
        }
        it++;
    } while ((Norme > Epsilon) && (it < it_max));

    return X_k_1;
}

// - - Gauss-Seidel

double *Gauss_Seidel(double **A, double *B, int Taille)
{
    // Allocation des vecteurs.
    double *X_k = Allocation_Vecteur(Taille);
    double *X_k_1 = Allocation_Vecteur(Taille);

    // Remplissage de X_k par des 1 (X_0).
    for (int i = 0; i < Taille; i++)
    {
        X_k[i] = 1;
    }

    double Norme;
    int it = 0;
    int it_max = 10;
    double Epsilon = 0.001;

    do
    {
        // Calcul de X_k_1.
        for (int i = 0; i < Taille; i++)
        {
            // Calcul de la somme des A[i][j] * X_k_1[j].
            double Somme = 0;
            for (int j = 0; j < i; j++)
            {
                Somme += A[i][j] * X_k_1[j];
            }
            // Calcul de la somme des a[i][j] * x_k[j].
            for (int j = i + 1; j < Taille; j++)
            {
                Somme += A[i][j] * X_k[j];
            }
            X_k_1[i] = (B[i] - Somme) / A[i][i];
```

```
        }

        // Calcul de la norme entre X_k_1 et X_k.
        Norme = 0;
        for (int i = 0; i < Taille; i++)
        {
            Norme += (X_k_1[i] - X_k[i]) * (X_k_1[i] - X_k[i]);
        }
        Norme = sqrt(Norme);

        // On remplace X_k par X_k_1.
        for (int i = 0; i < Taille; i++)
        {
            X_k[i] = X_k_1[i];
        }
        it++;
    } while ((Norme > Epsilon) && (it < it_max));

    return X_k_1;
}

// - Fonction principale.
int main()
{

    // Les différentes parties sont mises entre accolades afin que les
       variables aients une portée locale (propres à chaques parties).

    // - Méthodes directes

    // - - Méthode triangulaire inférieure.

    {
        // Définition de la taille.
        int Taille = 3;
        // Allocation des matrices et vecteurs.
        double **A = Allocation_Matrice(Taille);
        double *B = Allocation_Vecteur(Taille);

        // Remplissage des matrices et vecteurs.
        A[0][0] = 1;
        A[0][1] = 0;
        A[0][2] = 0;
        A[1][0] = 2;
        A[1][1] = 3;
        A[1][2] = 0;
        A[2][0] = 1;
        A[2][1] = 4;
        A[2][2] = -1;

        B[0] = 1;
        B[1] = 8;
        B[2] = 10;
```

```
    double *X = Sol_Inf(A, B, Taille);

    printf("La solution X de l'équation AX = B avec la méthode
        triangulaire inférieure est :\n");
    Afficher_Vecteur(X, Taille);

    // Désallocation des matrices et vecteurs.
    Desallocation_Matrice(A, Taille);
    Desallocation_Vecteur(B);
    Desallocation_Vecteur(X);
}

// - - Méthode triangulaire supérieure.

{
    // Définition de la taille.
    int Taille = 3;
    // Allocation des matrices et vecteurs.
    double **A = Allocation_Matrice(Taille);
    double *B = Allocation_Vecteur(Taille);

    // Remplissage des matrices et vecteurs.
    A[0][0] = 1;
    A[0][1] = 2;
    A[0][2] = 3;
    A[1][0] = 0;
    A[1][1] = 4;
    A[1][2] = 8;
    A[2][0] = 0;
    A[2][1] = 0;
    A[2][2] = 5;

    B[0] = 6;
    B[1] = 16;
    B[2] = 15;

    double *X = Sol_Sup(A, B, Taille);

    printf("La solution X de AX = B d'après la méthode triangulaire
        supérieure est :\n");
    Afficher_Vecteur(X, Taille);

    // Désallocation des matrices et vecteurs.
    Desallocation_Matrice(A, Taille);
    Desallocation_Vecteur(B);
    Desallocation_Vecteur(X);
}

// - - Elimination de Gauss.

{
    // Définition de la taille.
    int Taille = 3;
    // Allocation des matrices et vecteurs.
```

```
    double **A = Allocation_Matrice(Taille);
    double *B = Allocation_Vecteur(Taille);

    // Remplissage de A.
    A[0][0] = 1;
    A[0][1] = 2;
    A[0][2] = 3;
    A[1][0] = 5;
    A[1][1] = 2;
    A[1][2] = 1;
    A[2][0] = 3;
    A[2][1] = -1;
    A[2][2] = 1;

    // Remplissage de B.
    B[0] = 5;
    B[1] = 5;
    B[2] = 6;

    // Transformation de A en matrice triangulaire supérieure.
    Gauss(A, B, A, B, Taille);

    // Résolution de l'équation.
    double *X = Sol_Sup(A, B, Taille);

    // Affichage de X.
    printf("La solution X de AX = B d'après la méthode de l'
        élimination de Gauss est :\n");
    Afficher_Vecteur(X, Taille);

    // Désallocation des matrices et vecteurs.
    Desallocation_Matrice(A, Taille);
    Desallocation_Vecteur(B);
    Desallocation_Vecteur(X);
}

// - - Résolution par factorisation LU.

{
    // Définition de la taille.
    int Taille = 3;
    // Allocation des matrices et vecteurs.
    double **A = Allocation_Matrice(Taille);
    double *B = Allocation_Vecteur(Taille);

    A[0][0] = 1;
    A[0][1] = 2;
    A[0][2] = 3;
    A[1][0] = 5;
    A[1][1] = 2;
    A[1][2] = 1;
    A[2][0] = 3;
    A[2][1] = -1;
    A[2][2] = 1;
```

```
    B[0] = 5;
    B[1] = 5;
    B[2] = 6;

    double **L = Allocation_Matrice(Taille);

    // Factorisation de A = LU.
    LU(L, A, Taille);

    // Résolution de LY = B.
    double *Y = Sol_Inf(L, B, Taille);

    // Résolution de UX = Y.
    double *X = Sol_Sup(A, Y, Taille);

    // Affichage.
    printf("La solution X de AX = B d'après la méthode de la
        factorisation LU est : \n");
    Afficher_Vecteur(X, Taille);

    // Désallocation des matrices et vecteurs.
    Desallocation_Matrice(L, Taille);
    Desallocation_Matrice(A, Taille);
    Desallocation_Vecteur(B);
    Desallocation_Vecteur(Y);
    Desallocation_Vecteur(X);
}

// - - Cholesky.

{
    // Définition de la taille.
    int Taille = 4;
    // Allocation des matrices et vecteurs.
    double **A = Allocation_Matrice(Taille);
    double *B = Allocation_Vecteur(Taille);

    // Remplissage de A.
    A[0][0] = 1;
    A[0][1] = 1;
    A[0][2] = 1;
    A[0][3] = 1;
    A[1][0] = 1;
    A[1][1] = 5;
    A[1][2] = 5;
    A[1][3] = 5;
    A[2][0] = 1;
    A[2][1] = 5;
    A[2][2] = 14;
    A[2][3] = 14;
    A[3][0] = 1;
    A[3][1] = 5;
    A[3][2] = 14;
```

```
    A[3][3] = 15;

    // Remplissage de B.
    B[0] = 5;
    B[1] = 1;
    B[2] = 3;
    B[3] = 1;

    // Factorisation de A = L*L^T.
    double **L = Cholesky(A, Taille);

    // Résolution de LY = B.
    double *Y = Sol_Inf(L, B, Taille);

    // Transposition de L en L^T.
    Transposer(L, Taille);

    // Résolution de L^T*X = Y.
    double *X = Sol_Sup(L, Y, Taille);

    // Affichage de la solution X.
    printf("La solution X de AX = B d'après la méthode de Cholesky est
        : \n");
    Afficher_Vecteur(X, Taille);

    // Désallocation des matrices et vecteurs.
    Desallocation_Matrice(A, Taille);
    Desallocation_Matrice(L, Taille);
    Desallocation_Vecteur(Y);
    Desallocation_Vecteur(B);
    Desallocation_Vecteur(X);
}

// - Méthodes itératives.

// - - Jacobi.

{
    // Définition de la taille
    int Taille = 3;
    // Allocation des matrices et vecteurs.
    double **A = Allocation_Matrice(Taille);
    double *B = Allocation_Vecteur(Taille);

    // Remplissage de A
    A[0][0] = 4;
    A[0][1] = 1;
    A[0][2] = 1;
    A[1][0] = 1;
    A[1][1] = 3;
    A[1][2] = 1;
    A[2][0] = 2;
    A[2][1] = 0;
    A[2][2] = 5;
```

```
    // Remplissage de B
    B[0] = 1;
    B[1] = 1;
    B[2] = 1;

    // Résolution pour X.
    double *X = Jacobi(A, B, Taille);

    // Affichage de la solution X.
    printf("La solution X de AX = B avec la méthode Jacobi est : \n");
    Afficher_Vecteur(X, Taille);

    // Désallocation des matrices et vecteurs.
    Desallocation_Matrice(A, Taille);
    Desallocation_Vecteur(B);
    Desallocation_Vecteur(X);
}

// - - Gauss-Seidel.

{
    // Définition de la taille
    int Taille = 3;

    // Allocation des matrices et vecteurs.
    double **A = Allocation_Matrice(Taille);
    double *B = Allocation_Vecteur(Taille);

    // Remplissage de A (matrice quelconque).
    A[0][0] = 10;
    A[0][1] = 1;
    A[0][2] = 1;
    A[1][0] = 2;
    A[1][1] = 10;
    A[1][2] = 1;
    A[2][0] = 2;
    A[2][1] = 2;
    A[2][2] = 10;

    // Remplissage de B.
    B[0] = 12;
    B[1] = 13;
    B[2] = 14;

    // Résolution pour X.
    double *X = Gauss_Seidel(A, B, Taille);

    // Affichage de la solution X.
    printf("La solution X de AX = B avec la méthode de Gauss-Seidel : 
        \n");
    Afficher_Vecteur(X, Taille);

    // Désallocation des matrices et vecteurs.
```

```
        Desallocation_Matrice(A, Taille);
        Desallocation_Vecteur(B);
        Desallocation_Vecteur(X);
    }

    return 0;
}
```