INSA Rouen Normandie

Génie Mathématique 3ème année

Projet MMSN

Mars 2024

Algorithme Numérique

Étude et contrôle des erreurs sur la méthode de Gauss

ANKOUDY Yassir
DUQUESNOY Samuel

Encadrant
Bernard Gleyse

Table des matières

1 Introduction

La méthode de Gauss, aussi connue sous le nom d'élimination de Gauss, est une pierre angulaire des algorithmes de l'algèbre linéaire pour résoudre des systèmes d'équations linéaires. Son développement remonte à Carl Friedrich Gauss (1777-1855), un mathématicien allemand dont les contributions à la théorie des nombres, à la statistique, à l'analyse, à la géodésie, à l'astronomie et bien d'autres domaines sont inestimables. Bien que l'élimination des variables dans les systèmes linéaires fût connue avant Gauss, c'est lui qui a formalisé la méthode dans le cadre de ses recherches en astronomie, notamment pour prédire l'orbite des astéroïdes.

La méthode de Gauss transforme un système d'équations linéaires en un système équivalent plus simple à résoudre par substitution successive. Cette méthode est essentielle non seulement pour sa capacité à résoudre des systèmes d'équations, mais aussi pour sa pertinence dans l'analyse numérique moderne, notamment dans l'optimisation des calculs matriciels sur des ordinateurs.

Cependant, l'implémentation numérique de cette méthode n'est pas exempte de défis, notamment la propagation d'erreurs dues à la précision finie des calculs informatiques. Ces erreurs de calcul peuvent entraîner des divergences significatives entre les résultats théoriques et ceux obtenus par simulation. Depuis les travaux de Gauss, beaucoup de recherches ont été menées pour comprendre et améliorer la précision des algorithmes numériques. Des théoriciens comme M. Pichat et J. Vignes ont exploré l'ingénierie du contrôle de la précision des calculs, tandis que des mathématiciens tels que Nicholas Higham ont approfondi l'étude de la stabilité et de l'exactitude des algorithmes numériques.

Dans notre projet, nous aborderons la méthode de Gauss dans le contexte de la résolution numérique de systèmes linéaires, en mettant un accent particulier sur les phénomènes de propagation d'erreurs. Nous explorerons comment différentes stratégies de pivotage et de conditionnement préalable peuvent influencer la précision des résultats, et nous utiliserons des outils de diagnostic modernes tels que le débogueur GDB pour analyser les erreurs au niveau du code. Notre objectif est de combiner une compréhension historique et théorique approfondie avec des compétences pratiques en programmation pour améliorer la fiabilité des calculs scientifiques.

Ce travail s'inscrit dans une longue tradition de recherche en analyse numérique, où l'histoire et les développements modernes se rencontrent pour relever les défis posés par les limites de la computation numérique. En explorant les travaux fondamentaux et les avancées contemporaines, nous espérons non seulement résoudre efficacement des systèmes linéaires mais aussi contribuer à la compréhension globale de la propagation d'erreurs en computation numérique.

2 Présentation du problème

Dans ce projet, nous explorons la résolution de systèmes d'équations linéaires en utilisant la méthode de Gauss, une technique fondamentale en algèbre linéaire pour résoudre des systèmes matriciels. Cette méthode est non seulement cruciale pour comprendre les principes de l'algèbre linéaire mais sert également de base à de nombreuses applications en science et en ingénierie. Plus précisément, nous nous intéressons à la décomposition LU de la matrice, lors d'une étape de triangulisation, permettant ainsi la transformation du système initial en un format plus simple à résoudre.

La méthode de Gauss, ou élimination de Gauss, est souvent choisie pour sa robustesse et son efficacité dans le traitement de grands systèmes. Cependant, malgré ses avantages, la méthode n'est pas exempte d'erreurs potentielles, particulièrement en ce qui concerne les erreurs de calcul dues aux limitations de précision des ordinateurs. Ces erreurs peuvent influencer significativement la fiabilité des solutions obtenues, particulièrement dans les cas où la stabilité numérique est compromise par des matrices mal conditionnées ou des opérations arithmétiques susceptibles d'introduire des erreurs d'arrondi significatives.

Pour aborder ces questions, nous avons choisi de coder cette méthode en langage C, un choix motivé par la performance et le contrôle bas niveau offert par ce langage, permettant une analyse précise des opérations matricielles. Une partie essentielle de notre étude se concentre sur l'observation des erreurs locales qui apparaissent lors du calcul de chaque terme de la matrice transformée par la méthode de Gauss. L'analyse de ces erreurs se fait de manière indépendante pour chaque élément matriciel, ce qui nous aide à isoler et comprendre les erreurs spécifiques introduites lors des calculs.

Pour examiner ces erreurs plus en détail, nous utilisons GDB, le débogueur GNU. GDB nous permet non seulement de suivre pas à pas l'exécution du programme mais aussi d'inspecter l'état des variables et de la mémoire à tout moment du processus de calcul. Cette capacité est indispensable pour déterminer précisément où et pourquoi les erreurs numériques se manifestent. En comprenant ces dynamiques, nous pouvons envisager des améliorations algorithmiques ou des ajustements dans la manipulation des données qui réduisent l'impact de ces erreurs sur les résultats finaux.

3 Présentation générale de la méthode de Gauss

Considérons le système linéaire $AX = B$:

$$\begin{pmatrix} a_{11} & \cdots & \cdots & a_{1n} \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ a_{n1} & \cdots & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ b_n \end{pmatrix}$$

3.1 Décomposition LU

La décomposition LU d'une matrice utilisée dans la résolution de systèmes linéaires $AX = B$ est une méthode directe qui vise à décomposer une matrice A en le produit de deux matrices L (triangulaire inférieure) et U (triangulaire supérieure). L'objectif est de trouver les coefficients de L et U qui satisfont :

$$L = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ L_{21} & 1 & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ L_{n1} & \cdots & L_{(n-1)(n-1)} & 1 \end{pmatrix}, \quad U = \begin{pmatrix} U_{11} & \cdots & \cdots & U_{1n} \\ 0 & \ddots & & \vdots \\ \vdots & 0 & \ddots & \vdots \\ 0 & \cdots & 0 & U_{nn} \end{pmatrix}$$

Propriétés des matrices triangulaires

- **Proposition 1 :** Si L_1 et L_2 sont deux matrices triangulaires inférieures, alors $L = L_1 L_2$ est également triangulaire inférieure.
- **Proposition 2 :** Si L est une matrice triangulaire inférieure et inversible, alors L^{-1} est également triangulaire inférieure.
- **Corollaire :** Si U_1 et U_2 sont deux matrices triangulaires supérieures, alors $U = U_1 U_2$ est aussi triangulaire supérieure. De plus, si U est inversible, alors U^{-1} est également triangulaire supérieure.

3.2 Algorithme de descente

Algorithm 1 Algorithme de descente pour résoudre $Ly = b$

```
for i = 1 to n do
    x_i ← b_i
    for j = 1 to i − 1 do
        x_i ← x_i − a_ij x_j
    end for
    x_i ← x_i / a_ii
end for
```

3.3 Algorithme de remontée

3.4 Méthode de Gauss

Cette méthode nous permet de résoudre un système linéaire sous la forme $AX = B$ en utilisant au préalabre la factorisation LU. La méthode de Gauss nécéssite la décom-

Algorithm 2 Algorithme de remontée pour résoudre $Ux = y$

for $i = n$ **downto** 1 **do**
  $x_i \leftarrow y_i$
  for $j = i + 1$ **to** n **do**
    $x_i \leftarrow x_i - a_{ij}x_j$
  end for
  $x_i \leftarrow x_i/a_{ii}$
end for

position de la matrice A sous la forme LU, avec L une matrice triangulaire inférieure à diagonale unité et U une matrice triangulaire supérieure. Si cette décomposition LU de la matrice A existe, elle est alors unique, de plus toutes ces conditions sont vérifiées si et seulement si toutes les sous-matrices principales de A sont inversibles.

À présent, il faut résoudre le nouveau système qui est $LUX = B$. En posant $Y = UX$, on pourra résoudre d'abord le système $LY = B$ avec l'algorithme de descente, et par la suite, on résoudra $UX = Y$ avec l'algorithme de remontée. Dans notre cas, il permet de resoudre le système $Ux = y$. On procède de la même facon que le programme précédent. Ainsi, pour effectuer la méthode de Gauss, nous allons chercher une décomposition LU où L est triangulaire inférieure à diagonale unité et U est triangulaire supérieure. Tout d'abord, si cette décomposition $A = LU$ existe, alors elle est unique, d'après une propriété du cours. De plus, un théorème démontré durant le cours d'Analyse Numérique nous indique que toutes ces conditions $2n^3/3$ sont vérifiées si et seulement si toutes les sous-matrices principales de A sont inversibles.

3.4.1 Triangularisation :

L'étape essentielle dans la méthode de Gauss est la triangularisation, elle permet d'effectuer les étapes de descente et remontée. Expliquons à l'aide de matrices les étapes de la triangularisation.

La triangularisation :$a^{(k)}_{k+1,k}$
soit A une matrice inversible avec $(det\,(A) \neq 0)$
on a :

$$A = A^{(1)} \curvearrowright A^{(2)} \curvearrowright ..... \curvearrowright A^{(n)} = \begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \end{pmatrix}$$

avec

$$A^{(k)} = \begin{pmatrix} 0 & . & . & . & . & . & * \\ 0 & . & . & . & . & . & * \\ 0 & 0 & . & . & . & . & * \\ 0 & 0 & 0 & a^{(k)}_{k+1,k} & . & . & * \\ 0 & 0 & 0 & . & . & . & * \\ 0 & 0 & 0 & . & . & . & * \\ 0 & 0 & 0 & a^{(k)}_{nk} & . & . & * \end{pmatrix}$$

en effet ,

$$A^{(k)} = \begin{pmatrix} a_{11}^{(k)} & . & . & . & . & . & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & & . & . & . & a_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(3)} & . & . & . & a_{3n}^{(3)} \\ 0 & 0 & 0 & . & . & . & . \\ 0 & 0 & 0 & a_{k,k}^{(k)} & . & . & a_{kn}^{(k)} \\ 0 & 0 & 0 & . & . & . & . \\ 0 & 0 & 0 & a_{nk}^{(k)} & * & * & * \end{pmatrix}$$

$A^{(k)} \curvearrowright A^{(k+1)}$: Il faut modifier les lignes $i = k+1, n$ de sorte à ce que les termes en dessous de la diagonale principale soient non nuls.

$a_{i,j}^{(k+1)} = a_{i,j}^{(k)} - \frac{a_{i,j}^{(k)}}{a_{kk}^{(k)}} a_{k,j}^{(k)}$ avec $i = k+1$ à n

$b_i^{(k+1)} = b_i^{(k)} - \frac{a_{i,j}^{(k)}}{a_{kk}^{(k)}} b_k^{(k)}$ avec $j = k$ à n

Il faut que $a_{kk}^{(k)} \neq 0$ ou il faut que $a_{kk}^{(k)}$ soit un pivot.

tout ceci est équivalent a :

$$A^{(k+1)} = L^{(k)} A^{(k)}$$

$$b^{(k+1)} = L^{(k)} b^{(k)}$$

par suite on obtient :

$$L^{(k)} = \begin{pmatrix} 1 & 0 & . & . & . & . & 0 & 0 \\ 0 & 1 & 0 & & & . & . & 0 \\ . & & . & 0 & 0 & & & . \\ . & & & \frac{-a_{5k}^{(k)}}{a_{kk}^{(k)}} & 0 & 0 & & . \\ . & & & . & . & 0 & & . \\ . & & & & . & & 0 & . \\ . & & & & . & . & . & 0 \\ . & & & \frac{-a_{nk}^{(k)}}{a_{kk}^{(k)}} & & . & & 1 \end{pmatrix}$$

donc

$$A^{(n)} = L^{(n-1)}(.....L^{(2)} L^{(1)} A^{(1)}) \in UT$$

donc

$$U = A^{(n)} = \left(L^{(n-1)}.....L^{(1)}\right) A$$

$$\Longmapsto \left(L^{(n-1)}.....L^{(1)}\right)^{-1} U$$

Pour effectuer cette triangularisation, il est nécessaire d'avoir chaque coefficient $a_{kk} \neq 0$. Afin de vérifier cette condition, on peut utiliser, si besoin, un pivotage sur les lignes ou les colonnes (ou les deux) de la matrice A. Permutation des lignes : on permute les lignes k et p. Permutation des colonnes : On permute les colonnes k et p. Pivotage total. Finalement, la complexité de cette méthode de Gauss est de $2n^3/3$.

4 Résolution informatique du problème

4.1 Algorithme de Gauss avec Triangulisation et Résolution

Voici l'algorithme écrit en pseudo-code de la méthode de Gauss, incluant les parties de triangulisation et de résolution.

Triangulisation

- **Pour** $r = 1$ **à** $n - 1$ **faire**
  - **Pour** $i = r + 1$ **à** n **faire**
    - $l_{ir}^{(r)} = \frac{a_{ir}^{(r)}}{a_{rr}^{(r)}}$
    - $b_i^{(r+1)} = b_i^{(r)} - l_{ir}^{(r)} \times b_r^{(r)}$
    - **Pour** $j = r + 1$ **à** n **faire**
    - $a_{ij}^{(r+1)} = a_{ij}^{(r)} - l_{ir}^{(r)} \times a_{rj}^{(r)}$
    - $a_i^{(r+1)} = 0$

Résolution

- $x_n = \frac{b_n^{(n)}}{a_{nn}^{(n)}}$
- **Pour** $i = n - 1$ **à** 1 **faire**
  - $x_i = \frac{b_i^{(n)} - \sum_{j=i+1}^{n} a_{ij}^{(n)} x_j}{a_{ii}^{(n)}}$

4.1.1 Complexité du Programme

La complexité de ce programme est de $\mathcal{O}(n^3)$. En effet, pour la triangulisation, nous avons :

- $\frac{n(n-1)}{2}$ divisions,
- $\frac{n^3-n}{3}$ multiplications,
- $\frac{n^3-n}{3}$ additions,
- Soit un coût de $\frac{2n^3}{3}$, donc une complexité de $\mathcal{O}(n^3)$.

Pour la résolution, nous avons :

- $\frac{n(n-1)}{2}$ additions,
- $\frac{n(n-1)}{2}$ multiplications,
- Ce qui nous donne un coût total de $\frac{2n^3}{3}$, donc une complexité de $\mathcal{O}(n^3)$.

4.2 Codes en C pour la méthode de Gauss

On utilisera le language C pour implémenter cet algorithme. Il est important de préciser que tout notre projet est réalisé en double précision.

4.2.1 Fonction triangularisation :

```
void triangularisation(double **A, double *B, int N)
{
    int i,r,j;
    double**l=malloc(N*sizeof(double*));
    for(i=0;i<N;i++)
    {
    l[i]=malloc(N*sizeof(double));
    }

    for(r = 0; r < N-1; r++)
    {
        for(i = r+1; i < N; i++)
        {
            l[i][r] = A[i][r] / A[r][r];
            B[i]=B[i]-l[i][r]*B[r];
            for(j=r+1;j<N;j++)
            {
                A[i][j] = A[i][j] - l[i][r]*A[r][j];
                A[i][r] = 0;
            }
        }
    }
}
```

4.2.2 Fonction résolution :

```
void resolution(double **A, double *B, double *X, int N)
{
    int i, j;
    double somme;
    X[N-2]=B[N-2]/A[N-2][N-2];
    for(i=N-1;i>=0;i--)
    {
        somme=0;
        for(j=i+1;j<N;j++)
        {
            somme=somme+A[i][j]*X[j];
        }
        X[i]=(B[i]-somme)/A[i][i];
    }
}
```

5 Utilisation de GDB

Dans le cadre de notre projet, nous utiliserons GDB pour surveiller de près les manipulations matricielles et analyser la propagation des erreurs lors des opérations élémentaires sur les matrices. Ce processus implique l'affichage des valeurs des matrices A et L, tant en format décimal qu'en format binaire, ce qui est crucial pour comprendre comment les calculs à précision finie affectent les résultats de nos algorithmes numériques.

5.1 Affichage des Valeurs avec GDB

Pour observer les valeurs au sein de GDB, nous recourrons à la commande `print` pour afficher les contenus des variables. Voici un exemple de commande pour afficher la valeur d'une variable en format décimal :

```
(gdb) print A[i][j]
```

Pour examiner les mêmes valeurs en représentation binaire, GDB offre des fonctionnalités permettant de visualiser comment les données sont réellement stockées en mémoire. Ainsi, nous pouvons émettre une commande telle que :

```
(gdb) x/tg &A[i][j]
```

où `4wb` signifie que nous voulons examiner quatre mots (w pour word) en format binaire (b pour binary) à partir de l'adresse mémoire de l'élément $A[i][j]$.

Voici par exemple, un affichage obtenu lors de notre utilisation de GDB.

```
Breakpoint 1, triangularisation (A=0x5555555592a0, B=0x555555559390, N=4) at testLU.c:40
40                  b_av = B[j];
i = 0, j = 1
A[1][0] = 7.0000000000000000000
0x555555559300: 0100000000111000000000000000000000000000000000000000000000000000
A[0][0] = 10.000000000000000000
0x5555555592d0: 0100000000100100000000000000000000000000000000000000000000000000
L[1][0] = 0.69999999999999955591
0x555555559450: 0011111111100110011001100110011001100110011001100110011001100110

Breakpoint 1, triangularisation (A=0x5555555592a0, B=0x555555559390, N=4) at testLU.c:40
40                  b_av = B[j];
i = 0, j = 2
A[2][0] = 8.0000000000000000000
0x555555559330: 0100000000100000000000000000000000000000000000000000000000000000
A[0][0] = 10.000000000000000000
0x5555555592d0: 0100000000100100000000000000000000000000000000000000000000000000
L[2][0] = 0.80000000000000044409
0x555555559480: 0011111111101001100110011001100110011001100110011001100110011010

Breakpoint 1, triangularisation (A=0x5555555592a0, B=0x555555559390, N=4) at testLU.c:40
40                  b_av = B[j];
i = 0, j = 3
A[3][0] = 7.0000000000000000000
0x555555559360: 0100000000111000000000000000000000000000000000000000000000000000
A[0][0] = 10.000000000000000000
0x5555555592d0: 0100000000100100000000000000000000000000000000000000000000000000
L[3][0] = 0.69999999999999955591
0x5555555594b0: 0011111111100110011001100110011001100110011001100110011001100110
```

FIGURE 1 – Affichage des aj1 divisés par a11 en décimal puis en binaire

5.2 Automatisation avec GDB

Pour accroître l'efficacité de notre travail de débogage, nous utiliserons l'option `command` de GDB, qui nous permet de définir une série de commandes à exécuter à chaque point d'arrêt. Cela s'avère particulièrement utile pour répéter des tâches de débogage sans avoir à ressaisir les mêmes commandes. Voici un exemple :

```
(gdb) break triangularisation
(gdb) command
Type commands for breakpoint(s), one per line.
End with a line saying just "end".
>print A[i][j]
>x/tg &A[i][j]
>continue
>end
```

Avec ce script, à chaque fois que le point d'arrêt `triangularisation` est atteint, GDB affichera automatiquement la valeur courante de $A[i][j]$ en format décimal et binaire, puis continuera l'exécution jusqu'au prochain point d'arrêt.

Voici une `command` utilisée pour un affichage plus optimisé.

```
(gdb) break testLU.c:47
Breakpoint 2 at 0x5555555554d4: file testLU.c, line 47.
(gdb) command
Type commands for breakpoint(s) 2, one per line.
End with a line saying just "end".
>printf "i = %d, j = %d, k = %d\n", i, j, k
>printf "a_av = A[%d][%d] = %.21f\n", j, k, a_av
>x/tg &a_av
>printf "prod = L[%d][%d]*A[%d][%d] = %.21f\n",j,i,i,k, prod
>x/tg &prod
>printf "res = A[%d][%d] = %.21f\n", j, k, res
>x/tg &res
>continue
>end
(gdb) run
```

FIGURE 2 – Commande utilisée pour afficher a3k - L32 * A2k

Et voici le résultat obtenu :

FIGURE 3 – Affichage de la différence entre a3k et L32 * A2k

L'utilisation de GDB nous permettra non seulement de suivre les opérations élémentaires sur les matrices et de détecter les sources potentielles d'erreurs numériques, mais aussi d'optimiser notre flux de travail de débogage grâce à des outils d'automatisation efficaces.

6 Informations importantes concernant l'étude

6.1 Initialisation du Système Linéaire

Tout au long de ce projet, notre focus a été mis sur la résolution du système linéaire défini par l'équation $A\mathbf{x} = \mathbf{b}$. Ici, A désigne la matrice de Wilson, une matrice 4×4, initialisée de la manière suivante :

$$A = \begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix}$$

Une caractéristique intéressante de notre configuration est la manière dont nous avons initialisé le vecteur $\mathbf{b}$. Pour garantir que la solution du système, $\mathbf{x}$, soit un vecteur dont toutes les composantes sont égales à 1, nous avons choisi de définir $\mathbf{b}$ comme étant le vecteur des sommes des colonnes de la matrice A. Mathématiquement, cela se traduit par :

$$\mathbf{b} = A \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Ce qui donne explicitement :

$$\mathbf{b} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix}$$

Cela signifie que chaque composante de $\mathbf{b}$ est la somme des éléments de sa colonne correspondante dans la matrice A. Ce choix garantit que la solution théorique du système $A\mathbf{x} = \mathbf{b}$, sous l'hypothèse que A est non singulière, est le vecteur $\mathbf{x} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$, ce qui est vérifié par l'équation :

$$A \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \mathbf{b}$$

Cette initialisation n'est pas seulement une simplification mathématique, elle permet également de tester la précision et la stabilité numérique des méthodes de résolution que nous employons, notamment en observant comment les erreurs de calcul peuvent affecter une solution théoriquement connue et simple.

6.2 Conditionnement

Conditionnement d'une matrice

▷ **Définition :** Soit $A \in M_{nn}(\mathcal{R})$ une matrice carrée inversible .
On appelle conditionnement d'une matrice A le réel

$$\chi(A) = N(A) N\left(A^{-1}\right)$$

Inégalité du conditionnement

▷ **Propriété :** Soit $A \in M_{nn}(R)$ inversible . Alors on a : $\chi(A) \geq 1$ et cette inégalité provient de :

$$1 = N(I) = N\left(AA^{-1}\right) \leq N(A) N\left(A^{-1}\right) = \chi(A)$$

6.3 Conditionnement de la Matrice de Wilson

6.3.1 Calcul du Conditionnement

Le conditionnement de A a été calculé en utilisant plusieurs normes via Python et le module Numpy, illustrant différentes mesures de sensibilité :

— **Norme 1** (maximum de la somme des valeurs absolues des éléments de chaque colonne) : $\kappa_1(A) = 4488$

— **Norme Euclidienne** (la plus grande valeur singulière) : $\kappa_2(A) = 2984$

— **Norme Infinie** (maximum de la somme des valeurs absolues des éléments de chaque ligne) : $\kappa_\infty(A) = 4488$

6.3.2 Interprétation des Résultats

Ces résultats montrent que la matrice de Wilson est significativement mal conditionnée, particulièrement lorsque mesurée par la norme 1 et la norme infinie. Cela implique que pour des erreurs même minimes dans les données d'entrée, les solutions du système linéaire associé pourraient présenter de grandes variations. La faible valeur du conditionnement en norme 2 suggère une moindre sensibilité dans le contexte de cette norme, mais elle reste significative.

6.3.3 Conséquences Pratiques

Dans la pratique, un mauvais conditionnement comme celui observé pour la matrice de Wilson exige l'utilisation de techniques numériques avancées pour minimiser l'impact des erreurs d'arrondi et d'autres imprécisions computationnelles, telles que l'utilisation de la précision double ou de méthodes itératives améliorées. De telles précautions sont essentielles pour garantir la fiabilité des solutions obtenues par des calculs numériques.

6.4 Conversion de Décimal en Binaire en Python

Dans le cadre de ce projet, il est nécessaire de convertir fréquemment des nombres décimaux en leur équivalent binaire. Pour ce faire, nous avons implémenté une fonction en Python qui facilite cette tâche. Cette fonction utilise la représentation binaire IEEE 754 pour les nombres à virgule flottante.

Voici le code de la fonction en Python :

```
import struct

def float_to_bin(num):
    """ Converts a float to its binary representation according to IEEE 754.. """
    # Assurez-vous que le num est bien un float
    num = float(num)
    # Convertir le nombre flottant en une chaine de caracteres binaires
    return ''.join(f'{c:08b}' for c in struct.pack('!d', num))

# Exemple d'utilisation de la fonction
binary_string = float_to_bin(12.5)
print(binary_string) # Affiche la representation binaire de 12.5
```

Cette fonction prend en entrée un nombre flottant, le convertit d'abord en float pour s'assurer de la compatibilité avec le module ‘struct‘, puis le convertit en une chaîne de caractères représentant la séquence binaire correspondant à la norme IEEE 754.

7 Analyse des Erreurs obtenues

7.1 Erreurs d'affectations

Au début de notre étude, nous nous sommes penchés sur la question de savoir si des erreurs de précision pouvaient survenir lors de l'affectation des valeurs dans la matrice de Wilson, que nous désignerons par A, et dans le vecteur $\mathbf{b}$.

Nous nous intéressons ici plus particulièrement aux erreurs de précision qui pourraient affecter l'affectation de la première ligne de la matrice A.

En utilisant la double précision, nous avons procédé à l'affichage des valeurs $A[0][j]$ pour chaque j appartenant à l'ensemble $[\![0, 3]\!]$. Nous avons constaté que les valeurs affectées étaient exactes, ne révélant ainsi aucune erreur de précision.

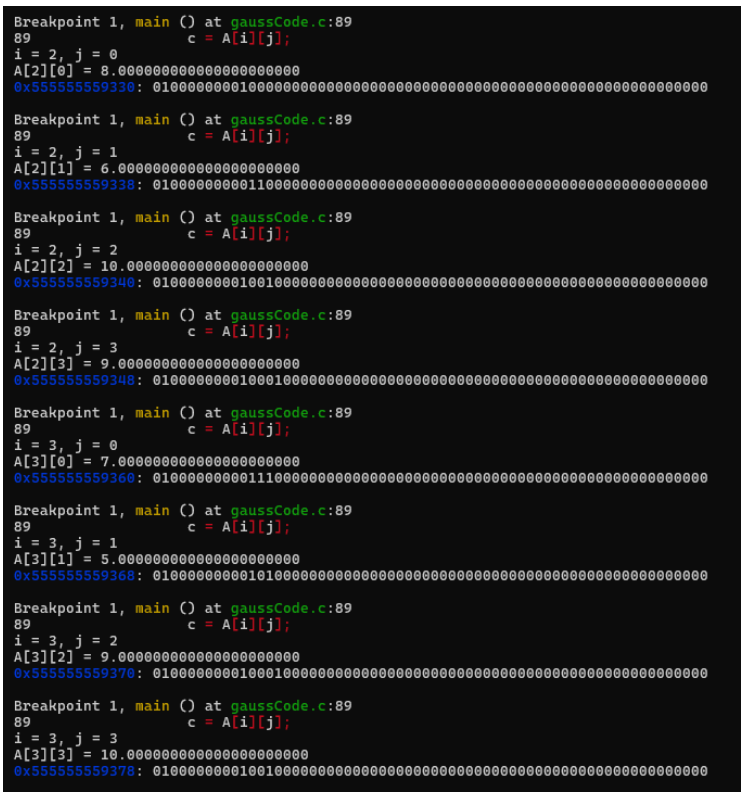

FIGURE 4 – Affichage des valeurs affectées à A[i][j] en décimal et binaire avec GDB

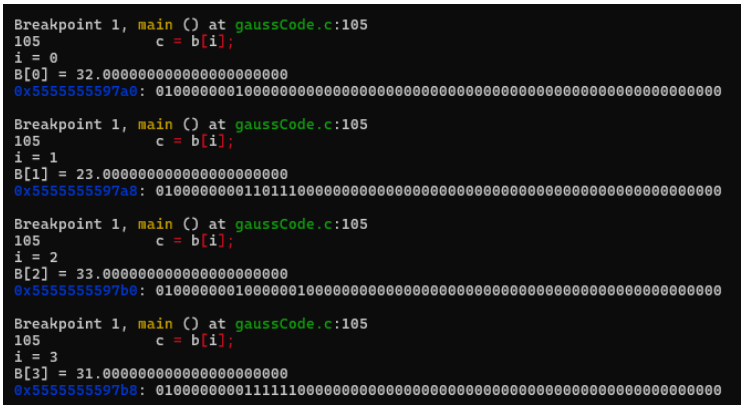

FIGURE 5 – Affichage des valeurs affectées à b[i] en décimal et binaire avec GDB

Cette constatation s'étend également aux autres lignes de la matrice A ainsi qu'au vecteur $\mathbf{b}$, où il a été observé que l'affectation se fait sans erreur notable.

7.2 Erreurs lors de la phase de triangularisation

Lors de la phase de triangularisation, des opérations élémentaires telles que la division pour calculer les coefficients de la matrice L et la soustraction pour actualiser les valeurs de la matrice A sont cruciales. Ces opérations sont exprimées comme suit :

$$L_{ji} = \frac{A_{ji}}{A_{ii}}$$

$$A_{jk} = A_{jk} - L_{ji} \cdot A_{ik}, \quad \text{pour } k \geq i$$

Voici un exemple d'un des résultats que nous avons obtenus pour la division :

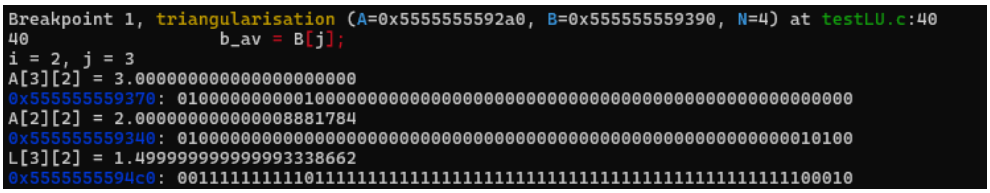

FIGURE 6 – Affichage des A[j][3]/A[3][3] en décimal et binaire avec GDB

Puis en exécutant le code Python (que vous retrouverez dans le même dossier cf. Readme), nous obtenons la valeur exacte de la division ainsi que l'erreur absolue et relative :

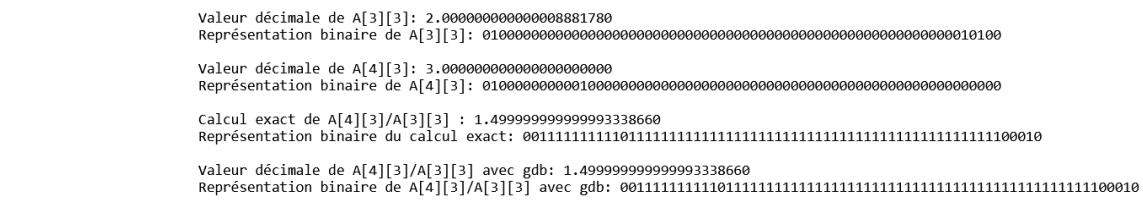

FIGURE 7 – Calcul de la valeur exacte de A[4][3]/A[3][3] avec Python

```
Erreur absolue de A[4][3]/A[3][3]: 1.0587911840678754238e-22
Erreur relative de A[4][3]/A[3][3]: 7.0586078937858675054e-23
```

FIGURE 8 – Affichage des erreurs absolue et relative pour la division de A[4][3] par A[3][3]

Au cours de nos observations, nous avons noté que même si la machine effectue ces calculs avec une grande précision, des erreurs de l'ordre de 10^{-20} sont présentes. Cela est dû à la représentation limitée des nombres en virgule flottante et aux inévitables erreurs d'arrondi qui surviennent lors de la manipulation de nombres très grands ou très petits. Voici un exemple de calcul effectué durant la phase de triangularisation et les erreurs correspondantes :

$$\text{Erreur absolue de } \frac{A[4][3]}{A[3][3]} : 1.0587911840678754238e-22,$$
$$\text{Erreur relative de } \frac{A[4][3]}{A[3][3]} : 7.0586078937858675054e-23.$$

Voici un autre exemple d'un des résultats que nous avons obtenus pour la soustraction :

```
Breakpoint 1, triangularisation (A=0x5555555592a0, B=0x555555559390, N=4) at testLU.c:47
47                      A[j][k] = res;
i = 2, j = 3, k = 2
a_av = A[3][2] = 3.0000000000000000000
0x7fffffffe0a0: 0100000000010000000000000000000000000000000000000000000000000000
prod = L[3][2]*A[2][2] = 3.0000000000000000000
0x7fffffffe098: 0100000000010000000000000000000000000000000000000000000000000000
res = A[3][2] = 0.0000000000000000000
0x7fffffffe0a8: 0000000000000000000000000000000000000000000000000000000000000000

Breakpoint 1, triangularisation (A=0x5555555592a0, B=0x555555559390, N=4) at testLU.c:47
47                      A[j][k] = res;
i = 2, j = 3, k = 3
a_av = A[3][3] = 5.0000000000000000000
0x7fffffffe0a8: 0100000000010100000000000000000000000000000000000000000000000000
prod = L[3][2]*A[2][3] = 4.4999999999999804600075
0x7fffffffe098: 0100000000010001111111111111111111111111111111111111111111101010
res = A[3][3] = 0.50000000000019539925
0x7fffffffe0a8: 0011111111100000000000000000000000000000000000000000000010110000
```

FIGURE 9 – Affichage des A[4][k] - L[4][3]*A[3][k] en décimal puis binaire avec GDB

Puis en exécutant le code Python (que vous retrouverez dans le même dossier cf. Readme), nous obtenons la valeur exacte de la division ainsi que l'erreur absolue et relative :

```
Valeur décimale de A[4][3]: 3.0000000000000000000
Représentation binaire de A[4][3]: 0100000000010000000000000000000000000000000000000000000000000000
Valeur décimale de L[4][3]*A[3][3]: 3.0000000000000000000
Représentation binaire de L[4][3]*A[3][3]: 0100000000010000000000000000000000000000000000000000000000000000
Valeur décimale de A[4][3] - L[4][3]*A[3][3] avec gdb: 0.0000000000000000000
Représentation binaire de A[4][3] - L[4][3]*A[3][3] avec gdb: 0000000000000000000000000000000000000000000000000000000000000000
--------------------------------------------------------------------------------
Valeur décimale de A[4][4]: 5.0000000000000000000
Représentation binaire de A[4][4]: 0100000000010100000000000000000000000000000000000000000000000000
Valeur décimale de L[4][3]*A[3][4]: 4.4999999999999980015986
Représentation binaire de L[4][3]*A[3][4]: 0100000000010001111111111111111111111111111111111111111111101010
Valeur décimale de A[4][4] - L[4][3]*A[3][4] avec gdb: 0.50000000000019539925
Représentation binaire de A[4][4] - L[4][3]*A[3][4] avec gdb: 0011111111100000000000000000000000000000000000000000000010110000
```

FIGURE 10 – Calcul des valeurs exactes des A[4][k] - L[4][3]*A[3][k] avec Python

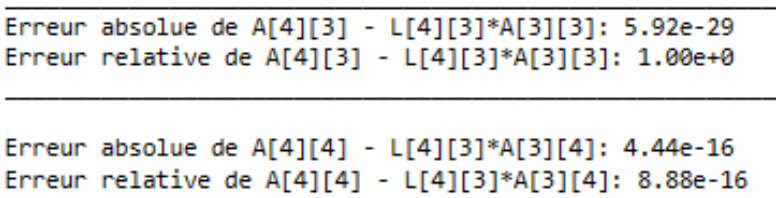

FIGURE 11 – Erreurs des A[4][k] - L[4][3]*A[3][k]

On remarque encore une fois que toutes les erreurs sont minimes. Ces erreurs, bien qu'extrêmement petites, peuvent s'accumuler et potentiellement affecter la stabilité du processus de résolution pour des systèmes linéaires de grande taille ou mal conditionnés.

7.3 Erreurs lors de la phase de descente

Une fois la matrice A transformée en une matrice triangulaire supérieure U, la descente est réalisée sur le vecteur B pour préparer la résolution. Pour $j > i$, nous mettons à jour B comme suit :

$$B_j = B_j - L_{ji} \cdot B_i$$

Voici un exemple d'un des résultats obtenus avec GDB :

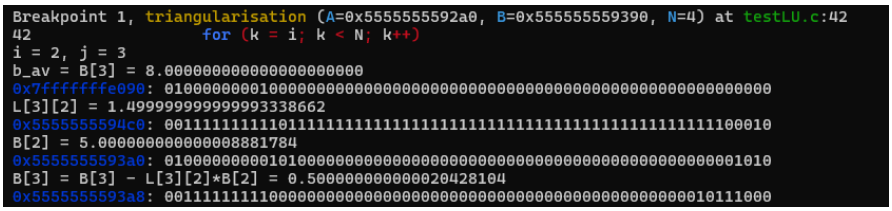

FIGURE 12 – Affichage B[j] - L[j][3]*B[3] en décimal puis binaire avec GDB

Puis en exécutant le code Python (que vous retrouverez dans le même dossier cf. Readme), nous obtenons la valeur exacte de la division ainsi que l'erreur absolue et relative :

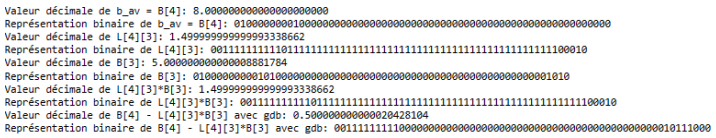

FIGURE 13 – Calcul de la valeur exacte B[j] - L[j][3]*B[3] avec Python

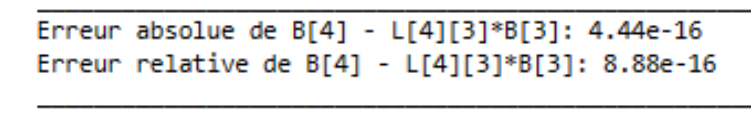

FIGURE 14 – Calcul des erreurs absolue et relative liées à B[j] - L[j][3]*B[3]

Durant notre étude, nous avons constaté que, bien que les calculs effectués par les ordinateurs soient d'une précision notable, des erreurs de l'ordre de 10^{-20} sont néanmoins détectables. Ces imprécisions résultent des limitations intrinsèques de la représentation des nombres à virgule flottante et des erreurs d'arrondi qui surviennent immanquablement lorsqu'on traite des nombres extrêmement grands ou petits.

7.4 Erreurs lors de la phase de remontée

La dernière étape de la méthode de Gauss est la remontée, qui résout le système triangulaire supérieur $UX = B$ pour obtenir la solution X. Le processus de remontée procède comme suit pour chaque élément X_i, en commençant par le dernier terme et en remontant :

```
Pour i allant de N-1 à 0 faire :
    somme ← 0
    Pour j allant de i+1 à N faire :
        somme ← somme + A[i][j] * X[j]
    Fin pour
    X[i] ← (B[i] - somme) / A[i][i]
Fin pour
```

Les erreurs peuvent se produire pendant cette phase en raison des opérations itératives de soustraction et de division, particulièrement si $A[i][i]$ est proche de zéro, ce qui peut conduire à une amplification des erreurs de calcul.

Voici un des résultats obtenus pour le produit de A[i][j] par X[j], étape nécéssaire dans cette phase :

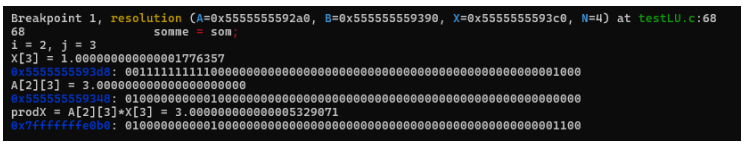

FIGURE 15 – Affichage de A[3][j]*X[j] en décimal puis binaire avec GDB

Puis en exécutant le code Python (que vous retrouverez dans le même dossier cf. Readme), nous obtenons la valeur exacte de la division ainsi que l'erreur absolue et relative :

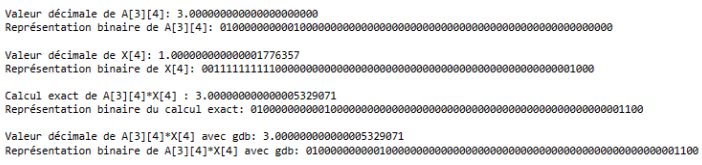

FIGURE 16 – Calcul de la valeur exacte des A[3][j]*X[j] avec Python

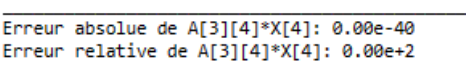

FIGURE 17 – Calcul des erreurs absolue et relative liées à A[3][j]*X[j]

Ici, les erreurs sont à 0 ce qui signifie que le calcul est exacte, mais vous pourrez aussi constater en visualisant les autres fichiers que dans cette phase, les erreurs sont les plus faibles.

Par la suite, nous allons nous intéresser à l'opération suivante : X[i] ← (B[i] - somme) / A[i][i], qui nous donne les X[i] (solutions du système) :

FIGURE 18 – Affichage des X[i] en décimal puis binaire avec GDB

Puis en exécutant le code Python (que vous retrouverez dans le même dossier cf. Readme), nous obtenons la valeur exacte de la division ainsi que l'erreur absolue et relative :

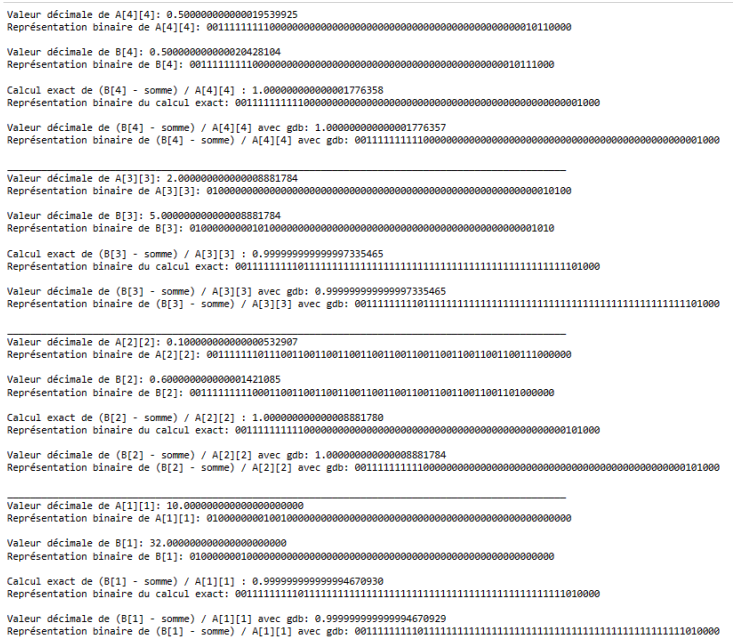

FIGURE 19 – Calcul de la valeur exacte des X[i] avec Python

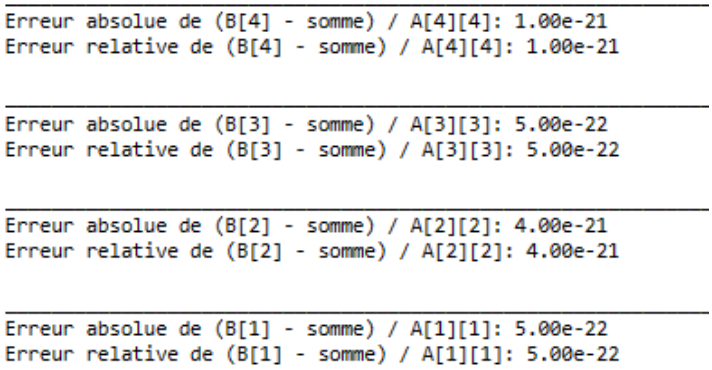

FIGURE 20 – Calcul des erreurs absolue et relative liées à X[i]

Ici, on remarque que les erreurs sont extrememement faibles de l'ordre de 10^{-20}, ce qui nous permet de dire que la solution de notre système linéaire est correcte, et que la méthode de Gauss est efficace, mais que toutefois de faibles erreurs persistent.

8 Conclusion

En conclusion de ce projet, l'exploration de la méthode de Gauss a révélé des enseignements notables sur les erreurs inhérentes aux différentes étapes de résolution d'un système linéaire. Lors de la triangularisation, nous avons constaté que, bien que les erreurs soient généralement minimes, elles peuvent s'accumuler, influençant les résultats des étapes subséquentes. Les phases de descente et de remontée, malgré leur robustesse, ne sont pas exemptes de ces erreurs relatives faibles, mais significatives lorsqu'elles se propagent à travers les calculs itératifs.

Cette étude a été l'occasion d'approfondir notre compréhension de GDB et de constater que, même avec une grande précision numérique, les calculs informatiques ne sont pas toujours parfaitement exacts. L'enregistrement des résultats intermédiaires en formats binaire et décimal pour les phases de descente et de remontée a été un outil précieux pour tracer et analyser ces erreurs.

Nous tenons à exprimer notre gratitude envers M. Gleyse pour l'assistance généreuse qu'il nous a accordée tout au long de ce projet, nous permettant d'approfondir notre compréhension théorique et pratique de la résolution numérique de systèmes linéaires.

Recherches concernant la méthode de Gauss

Cette technique, nommée en hommage à Carl Friedrich Gauss, était déjà connue des mathématiciens chinois dès le premier siècle après J.-C. Elle permet de résoudre des systèmes d'équations linéaires en utilisant ce que l'on appelle aujourd'hui l'élimination de Gauss-Jordan. Ce projet a impliqué l'application des concepts appris dans le cours d'analyse numérique du semestre 5, enseigné par Maria Kazakova. Pour plus d'informations, consultez la page Wikipedia sur l'élimination de Gauss-Jordan : `https://fr.wikipedia.org/wiki/%C3%89limination_de_Gauss-Jordan`.

Recherche concernant GDB

GDB, ou GNU Debugger, est un débogueur libre conçu par Richard Stallman en 1986, qui supporte plusieurs langages de programmation comme Fortran, C et C++. Il est particulièrement apprécié pour sa portabilité sur de nombreux systèmes de type Unix. Ce logiciel est extrêmement utile pour le débogage et l'analyse de programmes. Au cours de ce projet, nous avons appris à utiliser plusieurs de ses fonctionnalités, notamment la fonction *watch*. Pour plus d'informations sur GDB et son utilisation, consultez `http://sdz.tdct.org/sdz/deboguer-son-programme-avec-gdb.html` et la page Wikipedia du GNU Debugger : `https://fr.wikipedia.org/wiki/GNU_Debugger`.

Références

[1] D. E. Knuth, *The art of computer programming : seminumerical algorithms*, Addison Wesley, 2nd edition, 1981.

[2] M. Pichat, J. Vignes, *Ingénierie du contrôle de la précision des calculs sur ordinateur*, Technip, 1993.

[3] N. J. Higham, *Accuracy and stability of numerical algorithms*, SIAM, 1996.

[4] F. Chaitin-Chatelin, V. Frayssé, *Lectures on finite precision computations*, SIAM, 1996.

[5] C. Brézinski, *Algorithmes d'accélération de la convergence - étude numérique*, Technip, 1978.

[6] J. P. Demailly, *Analyse numérique et équations différentielles*, PUG, 1991.