Résolution de systèmes d'équations linéaires (méthode indirecte)

Valentin PORTAIL et Jérémie VILLEPREUX

20 octobre 2022

Table des matières

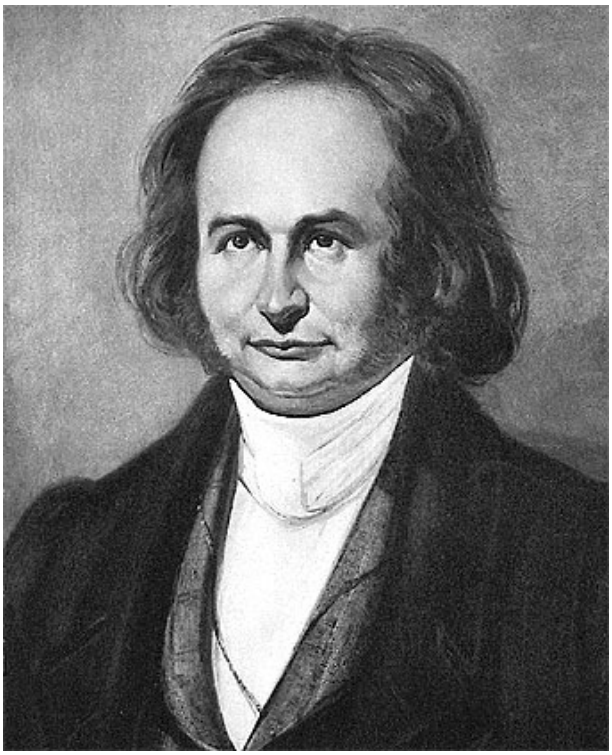



Figure 1 – Charles Gustave Jacob Jacobi et Philipp Ludwig von Seidel

1 Pré-requis

1.1 Les bibliothèques nécessaires

Ce projet nécessite l'installation des trois bibliothèques standards suivantes :
- `stdio.h`, pour la gestion des affichages et saisie clavier ;
- `stdlib.h`, pour la gestion des allocations dynamiques ;
- `math.h`, pour l'utilisation de la fonction puissance ;

1.2 Comment compiler notre projet ?

Un fichier `Makefile` est également disponible pour compiler le programme. Le compilateur choisi est `gcc` et l'exécutable généré aura pour nom `prog`. Il s'obtient avec la commande :

```
$ make
```

Il est également possible de compiler le programme sans utiliser le `Makefile` à disposition avec la commande :

```
$ gcc -Wall -Wextra -o prog *.c -lm
```

Remarque 1. La compilation de tous les fichiers sources est nécessaire à l'obtention de l'exécutable.

Remarque 2. Il y a également un répertoire sous le nom de `CSV` qui contient du code `Python`. Pour lancer le code, il est nécessaire d'avoir les librairies `mathplotlib.pyplot` et `numpy`. Mais l'exéctution n'est pas nécessaire (car déjà fait).

2 Rappel rapide des méthodes

2.1 Principe des méthodes de résolution indirectes

2.1.1 Contexte

Soient $A \in M_{n,n}(\mathbb{R})$ une matrice inversible et $b \in M_{n,1}(\mathbb{R})$ une matrice colonne. Dans ce document, notre objectif est de résoudre notre système $(Ax = b)$ non pas de manière exacte, mais de manière approchée et efficace, c'est-à-dire avec une complexité algorithmique plus faible et donc un temps d'exécution du programme plus rapide.

2.1.2 Méthode principale

Nous allons chercher des matrices M et $N \in M_{n,n}(\mathbb{R})$ telles que $A = M - N$ avec M facile à inverser. Notre système peut alors s'écrire de la manière suivante :

$$Ax = b \iff Mx = Nx + b \iff x = M^{-1}Nx + M^{-1}b$$

On pose $F(x) = M^{-1}Nx + M^{-1}b$. Nous pouvons en déduire un algorithme itératif pour trouver une solution approchée du système :

— Choisir un point initial $x^{(0)}$
— Pour $k \geq 0$, poser $x^{(k+1)} = F(x^{(k)}) = M^{-1}Nx^{(k)} + M^{-1}b$

Si la suite $(x^{(k)})_{k\in\mathbb{N}}$ converge, on obtiendra une solution approchée du système d'équation linéaire :

$$x^{(\infty)} = F(x^{(\infty)}) = M^{-1}Nx^{(\infty)} + M^{-1}b$$

À chaque étape, on calcule le résidu $r^{(k)} = ||Ax^{(k)} - b||$. Si $(x^{(k)})_{k\in\mathbb{N}}$ converge, alors notre algorithme s'arrêtera lorsque le résidu sera très petit, c'est-à-dire inférieur à une certaine valeur que l'on fixera (ici 10^{-6}). Ceci semble être un bon compromis entre une forte précision et un nombre d'itérations raisonnable.

Sinon, la suite diverge et l'algorithme s'arrêtera au bout d'un certain nombre d'itérations de k que l'on fixera aussi afin d'éviter une boucle infinie (ici 512). Ici, on a choisi arbitrairement 512, car pour les matrices creuses, la convergence est bien plus lente, et donc un nombre d'itération trop faible implique une mauvaise approximation du résultat. Nous reviendrons sur ce point dans la section 4 avec le cas potentiel des matrices creuses (*e.g.* A_6).

Nous pouvons également décomposer notre matrice A de la manière suivante :

$$A = D - E - F$$

où D est diagonale, E est triangulaire supérieure et F est triangulaire inférieure.

Nous allons voir deux méthodes indirectes de résolution de systèmes linéaires : la méthode de Jacobi et celle de Gauss-Seidel. Ces deux méthodes ont un fonctionnement très similaire. La principale différence réside dans la manière dont nous allons associer les matrices M, N avec les matrices D, E, F.

2.1.3 Aménagement du résidu

La formule du « *résidu* » donnée en TP diffère légèrement de celle donnée en cours. Nous utiliserons par la suite celle du TP. Par abus de langage, nous appellerons cela « résidu », mais il s'agit en réalité de « l'*erreur relative* ». La différence majeure entre les deux est que pour calculer l'*erreur relative*, il faut connaître la **solution exacte**. Alors que pour le *résidu* non. Néanmoins, ici, l'erreur relative est aussi précise que le résidu au bout d'un certain nombre d'itérations. En effet, le résidu commence avec des valeurs plus éloignées, mais il converge plus vite vers 0 que l'*erreur relative*. Pour rappel, on calcul l'*erreur relative* avec la formule :

$$r^k = \frac{Max_{i\in[\![1,n]\!]}|\bar{x}_i - \widetilde{x}_i^k|}{Max_{i\in[\![1,n]\!]}|\bar{x}_i|} = Max_{i\in[\![1,n]\!]}|\bar{x}_i - \widetilde{x}_i^k|$$

2.2 Méthode de Jacobi

Posons alors $M = D$ et $N = E + F$. Notre système s'écrit alors de la manière suivante à l'itération $k+1$:

$$Ax = b \iff Dx^{(k+1)} = (E+F)x^{(k)} + b$$

L'algorithme de JACOBI pour une précision ϵ est le suivant :

Algorithm 1: Méthode de Jacobi

Input : A : matrice de taille $n \times n$;
B : matrice de taille $n \times 1$;
n : entier désignant la taille de A et de B ;
$x^{(0)}$: matrice de taille $n \times 1$ représentant le point initial ;
ϵ : précision voulue pour la solution ;
k_{max} : nombre maximal d'itérations de l'algorithme ;

$\epsilon^{(0)} = ||Ax^{(0)} - b||$ // Ou la valeur du résidu de la section 2.1.3.
$k = 0$
while $\epsilon^{(k)} > \epsilon$ *and* $k < k_{max}$ **do**
  for $i = 1, \dots, n$ **do**
    $x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right]$
  end for
  $\epsilon^{(k+1)} = ||Ax^{(k+1)} - b||$ // idem.
  $k = k + 1$
end while

Remarque 3. On a $a_{ii} = d_{ii}$ et que $\sum_{j \neq i} a_{ij} x_j^{(k)} = \sum_{j=1}^{n} e_{ij} x_j^{(k)} + f_{ij} x_j^{(k)}$, avec d_{ij}, e_{ij} et f_{ij} les coefficients respectifs des matrices D, E, et F.

2.3 Méthode de Gauss-Seidel

On pose $M = D - E$ et $N = F$. Notre système s'écrit alors de la manière suivante à l'itération $k+1$:

$$Ax = b \iff Dx^{(k+1)} = Ex^{(k+1)} + Fx^{(k)} + b$$

L'algorithme de Gauss-Seidel pour une précision ϵ est le suivant :

Algorithm 2: Méthode de Gauss-Seidel

Input : A : matrice de taille $n \times n$;
B : matrice de taille $n \times 1$;
n : entier désignant la taille de A et de B ;
$x^{(0)}$: matrice de taille $n \times 1$ représentant le point initial ;
ϵ : précision voulue pour la solution ;
k_{max} : nombre maximal d'itérations de l'algorithme ;

$\epsilon^{(0)} = ||Ax^{(0)} - b||$ // Ou la valeur du résidu de la section 2.1.3.
$k = 0$
while $\epsilon^{(k)} > \epsilon$ *and* $k < k_{max}$ **do**
  for $i = 1, \dots, n$ **do**
    $x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right]$
  end for
  $\epsilon^{(k+1)} = ||Ax^{(k+1)} - b||$ // idem.
  $k = k + 1$
end while

Remarque 4. On a $a_{ii} = d_{ii}$, que $\sum_{j<i} a_{ij} x_j^{(k+1)} = \sum_{j=1}^{n} e_{ij} x_j^{(k+1)}$ et que $\sum_{j>i} a_{ij} x_j^{(k+1)} = \sum_{j=1}^{n} f_{ij} x_j^{(k)}$, avec d_{ij}, e_{ij} et f_{ij} les coefficients respectifs des matrices D, E, et F.

3 Présentation des programmes commentés

3.1 Structure générale du programme

Le programme est divisé en trois dossiers (le répertoire courant, le dossier `Documentation` et `CSV`). Le répertoire courant contient principalement les fichiers `.c`, le dossier `Documentation` contient la documentation complémentaire du projet, et le dossier `CSV` contient la fonction pour tracer les graphes en `Python`, et les résultats obtenus ainsi que les fichiers `CSV` de départ.

Revenons plus en détails sur chacun des trois dossiers. Tout d'abord, le répertoire courant :

- `methodeJ_Gs.c` : contient l'implémentation de la méthode de Jacobi et de Gauss-Seidel, diagonale strictement dominante, ... ;
- `MatriceTest.c` : contient principalement l'implémentation des matrices de test, affichage, libération de mémoire, ... ;
- `main.c` : contient le programme principal faisant appel aux autres fichiers ;
- `graph_csv.py` : contient ce qui est nécessaire à la réalisation des tracés `Python` ;

Remarque 5. À chaque fichier `.c` correspond un fichier `.h`, qui contient essentiellement les signatures des fonctions (sauf pour le `main.c`) la définition des variables de pré-processeur et des structures ;

3.2 Commentaire du programme

Pour en savoir plus sur nos fonctions, il est possible de se référer aux commentaires dans le code. Il est aussi possible de consulter une documentation plus détaillée. Elle se trouve dans le répertoire `Documentation` sous le nom de `documentationCplt.pdf`.

3.3 Choix d'implémentation

3.3.1 `matriceTest.c`

On a décidé de créer une fonction pour chaque matrice. Ce choix d'implémentation permet d'avoir un gain de place en mémoire. En effet, cela permet d'allouer la fonction uniquement si on a l'intention de s'en servir après, et non d'allouer toutes les fonctions au début du programme.

La coordonnée pour une ligne de la matrice b est obtenue en faisant la somme des coefficients de cette même ligne dans la matrice A. Nous avons décidé de calculer ces valeurs à la main, puis de rentrer le résultat directement et non de laisser la machine le calculer, afin d'éviter les potentielles erreurs d'arrondis. Cela permet donc d'éviter des approximations du résultat, avant même d'appliquer une des deux méthodes itératives.

3.3.2 `methodeJ_Gs.c`

Pour plus de détails, sur les méthodes, se référer aux algorithmes de la section 2.2 et 2.3. Notons également que ce fichier contient une fonction pour écrire dans un fichier `CSV` les valeurs des erreurs relatives. Et un algorithme pour savoir si une matrice A quelconque est à diagonale strictement dominante. Pour rappel, on dit que $A \in M_{n,n}(\mathbb{R})$ est à diagonale strictement dominante *si et seulement si* :

$$\forall i \in [\![1, n]\!], \quad |a_{i,i}| > \sum_{j \neq i} |a_{i,j}|$$

3.3.3 `main.c`

Plutôt que d'afficher toutes les matrices et la résolution associée à ces matrices, nous avons décidé de laisser le choix à l'utilisateur de sélectionner la matrice et la méthode itérative de son choix. On a implémenté ce « choix » à l'aide d'un `switch`. Les différents choix possibles sont rappelés grâce à un menu de sélection. Tout ceci se veut être le plus intuitif possible.

3.4 Quelques graphes

Voici également le graphe d'appel de nos fonctions en figure 2 ainsi que le graphe d'inclusion de nos librairies 3.

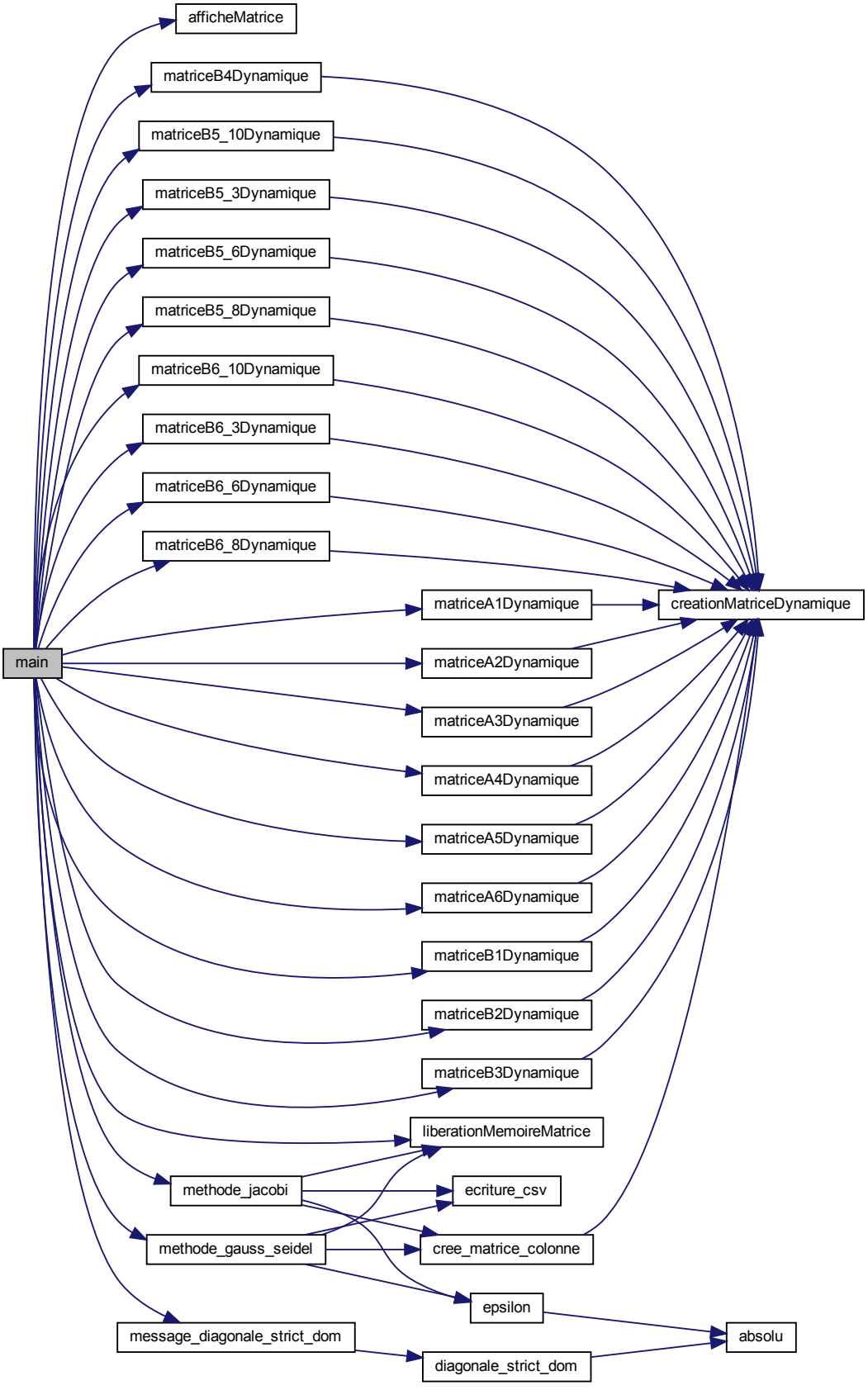


Figure 2 – Graphe d'appel des fonctions

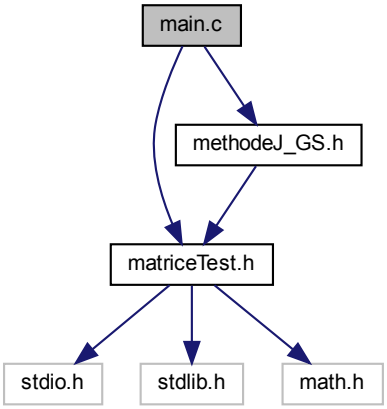


Figure 3 – Graphe d'inclusions de nos librairies

4 Commentaires sur les jeux d'essais

4.1 Implémentation des matrices de test

Le test des différentes matrices sera assez similaire à celui fait pour les méthodes directes (c.f. le rapport précédent).

On implémente dans le fichier `matriceTest.c` des matrices de test :

$$A_1 = \begin{pmatrix} 3 & 0 & 4 \\ 7 & 4 & 2 \\ -1 & 1 & 2 \end{pmatrix}, \; A_2 = \begin{pmatrix} -3 & 3 & 6 \\ -4 & 7 & 8 \\ 5 & 7 & -9 \end{pmatrix}, \; A_3 = \begin{pmatrix} 4 & 1 & 1 \\ 2 & -9 & 0 \\ 0 & -8 & 6 \end{pmatrix}, \; A_4 = \begin{pmatrix} 7 & 6 & 9 \\ 4 & 5 & -4 \\ -7 & -3 & 8 \end{pmatrix}$$

Les matrices A_5 et A_6 sont définies différemment. Au lieu de définir un à un tous les coefficients, ces derniers sont déterminés grâce à certaines conditions.

Ainsi, pour $k \in \{5, 6\}$, on a $A_k = (a_{ij})_{i,j \in [\![1,n]\!]}$ avec $n = 3, 6, 8, 10$ telle que :

$$A_5 = \begin{cases} a_{ii} = 1 \\ a_{1j} = a_{j1} = 2^{1-j} \\ 0 \text{ sinon} \end{cases} \quad \text{et } A_6 = \begin{cases} a_{ii} = 3 \\ a_{ij} = -1 \text{ si } j = i+1, i < n \\ a_{ij} = -2 \text{ si } j = i-1, i > 1 \\ 0 \text{ sinon} \end{cases}$$

On choisit les vecteurs b afin que la solution exacte du système d'équations soit : $\bar{x}_i = 1, \forall i \in [\![1, n]\!]$. Pour cela, le i-ième coefficient de la matrice b sera égal à la somme des coefficients sur la i-ième ligne de la matrice A. Autrement dit :

$$\forall i \in [\![1, n]\!], \;\; b_i = \sum_{j=1}^{n} a_{ij}$$

Enfin, pour toutes les matrices de test, on choisira le vecteur nul comme point initial $x^{(0)}$:

$$x^{(0)} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

A_i	$p(J)$	NbIter Jacobi	$p(GS)$	NbIter Gauss-Seidel	Convergence (?)
A_1	1.192983×10^{26}	512	1.756770×10^{102}	512	Diverge
A_2	3.901246×10^{77}	512	1.252239×10^{154}	512	Diverge
A_3	3.300182×10^{-7}	19	3.075388×10^{-7}	8	Converge
A_4	6.851836×10^{-7}	35	5.466671×10^{-7}	54	Converge
A_5	9.389577×10^{-7}	26	6.259694×10^{-7}	14	Converge
A_6	9.785808×10^{-7}	150	9.045951×10^{-7}	72	Converge

Figure 4 – Table de l'erreur relative en fonction des itérations selon la méthode

4.2 Résultat

4.2.1 Solution des systèmes

Nous avons testé les différentes matrices A_i avec $i \in 1, 2, 3, 4, 5, 6$. Pour chaque matrice et pour chaque méthode, on notera le nombre d'itérations effectuées, que l'on notera $NbIter$, et le résidu obtenu à la dernière étape, que l'on notera $p()$. Ce dernier résidu est suffisant pour déterminer si la méthode converge ou diverge.

Les matrices A_5 et A_6 peuvent avoir différentes tailles : 3, 6, 8 ou 10. Cependant, leur mode de construction reste similaire pour des tailles différentes. Ainsi, elles convergeront plus ou moins à la même vitesse. Il suffira donc de tester A_5 et A_6 avec pour taille $n = 10$.

Le résultat obtenu avec les deux méthodes itératives n'est pas toujours égal à 1, contrairement à la méthode de Gauss. Par exemple, pour la matrice A_6 de taille 10, on obtient comme matrice X :

$$X_6 = \begin{pmatrix} 1.000000 \\ 1.000000 \\ 1.000000 \\ 1.000000 \\ 0.999999 \\ 0.999999 \\ 0.999999 \\ 0.999999 \\ 0.999999 \\ 0.999999 \end{pmatrix}$$

Les méthodes itératives comme celle de Jacobi ou de Gauss-Seidel ne donnent donc pas des solutions exactes, mais plutôt des solutions approchées.

4.2.2 Renseignement obtenue grâce à la diagonale strictement dominante

Soit $A \in M_n(\mathbb{R})$. On sait que si A est définie positive, alors la méthode de Jacobi est définie. Malheureusement, il paraît difficile de vérifier que les matrices tests proposées le sont. En effet, il faudrait vérifier que :

$$\forall x \in M_{n,1}(\mathbb{R}), \quad x^T A x \geq 0$$

Or, il est impossible de vérifier ce "$\forall x$". En effet, la mémoire de l'ordinateur étant de taille finie, on ne peut pas représenter toutes les matrices x. Il suffirait alors de faire une preuve à la main, mais cela sort de notre cadre d'étude. Néanmoins, **si** A est définie positive et qu'elle est à diagonale strictement dominante, **alors** la méthode de Jacobi converge nécessairement.

De même, **si** A est symétrique et définie positive **alors** la méthode de Gauss-Seidel converge, mais là encore, nous avons le même problème. Il est facile de vérifier si une matrice est symétrique ou non, mais il reste difficile de savoir si elle est définie positive. D'après notre fonction `Diagonale_stric_dom`, seules les matrices A_5 et A_6 sont à diagonale strictement dominante.

Comme énoncé plus tôt, seules A_1 et A_2 divergent et non A_3 et A_4. Ceci s'explique simplement par le fait que ce sont des implications et non des équivalences. C'est-à-dire qu'on peut avoir des matrices convergentes ne vérifiant pas nécessairement les hypothèses ci-dessus. Le seul renseignement apporté est que si elle vérifie les hypothèses, alors elle converge obligatoirement.

Dans notre cas, il n'est donc pas intéressant de savoir si les matrices sont à diagonale strictement dominante car on applique dans tous les cas la méthode de Jacobi et Gauss-Seidel pour trouver une solution x ; Si les

valeurs de notre algorithme se terminent en un nombre d'itérations maximal fixé, alors il est fortement probable que les méthodes ne convergent pas. Dans tous les cas, il est très facile de vérifier cela en affichant les valeurs de x. Les valeurs changent beaucoup d'une itération à l'autre.

4.2.3 Comparaison et Jacobi d'un point de vue de la précision

Pour comparer l'erreur relative entre les méthodes de Jacobi et Gauss-Seidel, nous avons eu l'idée de réaliser des graphiques qui montrent l'évolution de l'erreur relative selon l'itération pour chaque matrice. Nous avons choisi une échelle logarithmique.

Cela s'explique par le fait que cette échelle permet d'avoir beaucoup de précision sur les petites valeurs et moins sur les grandes valeurs. En effet, il était intéressant de faire cela, car on s'est rendu compte que la vitesse de convergence des méthodes itératives était importante au début et qu'elles convergeaient moins vite après. L'échelle logarithmique montre sur un petit intervalle un grand nombre de valeurs.

La seconde raison est qu'il ne nous importe peu de savoir avec exactitude les valeurs des erreurs relatives pour les matrices test proposées pour chacune des deux méthodes et qu'il est plutôt intéressant de pouvoir les comparer. Pour cela, l'échelle logarithmique est une bonne idée. On obtient alors des graphes avec une échelle que l'on n'a pas l'habitude d'utiliser (car non-*linéaire*). Les graphiques sont disponibles en figure 5.

On voit que A_1 et A_2 divergent. Nous avons tronqué les graphes à 140 itérations, car on a jugé cela non-intéressant. En effet, on peut conjecturer que les droites se prolongent (vérifié en pratique jusqu'à 500 itérations, la droite se prolonge). Cela montre que la divergence des méthodes itératives est exponentielle. On peut alors se rendre très vite compte en pratique si les méthodes divergent ou non, car la différence des erreurs entre deux itérations n'est pas négligeable.

On voit également que pour toutes les matrices sauf A_4, l'erreur relative converge plus vite vers 0 avec la méthode de Gauss-Seidel qu'avec celle de Jacobi. Cela vérifie ce qui a été vu en cours : Gauss-Seidel converge plus vite que Jacobi. Néanmoins, ce n'est pas toujours le cas, la matrice A_4 en est un exemple. Nous n'avons pas d'explication à cela du fait de la « banalité » de cette matrice. On en conclut que Gauss-Seidel est plus efficace que Jacobi, car elle converge en moins d'itérations. De plus, la convergence est assez rapide. Seule A_6 ne converge pas aussi rapidement. Ceci s'explique par le fait qu'il s'agit d'une matrice *creuse*.

Remarque 6. Un petit plateau apparaît au delà de 10^{-6}. Ceci est spécifique à `Python`, car il manque de précision sur les valeurs plus faibles.

4.2.4 Comparaison du temps d'exécution des vitesses

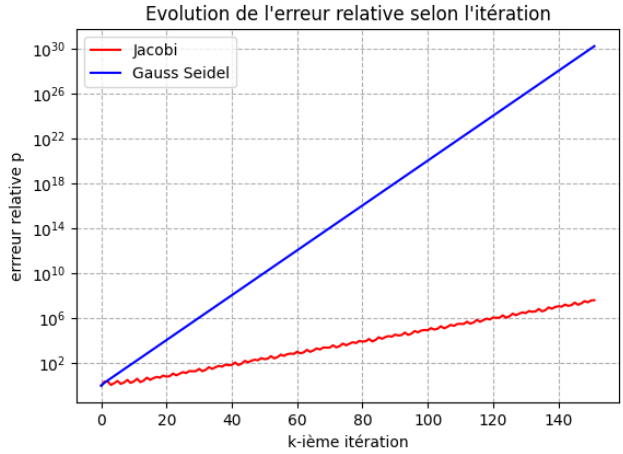
Nous avons également fait un programme pour calculer le temps d'exécution des trois méthodes vues, afin de déterminer laquelle est la plus rapide entre Gauss, Jacobi et Gauss-Seidel. Nous avons fait le test sur A_3, A_4 et A_6. Le choix de ces matrices n'est pas anodin. En effet, elles ont toutes les trois leurs propriétés. A_3 est une matrice « banale ». A_4, car on a vu que la méthode de Jacobi se termine en moins d'itérations que Gauss-Seidel. Et enfin A_6, car il s'agit d'une matrice creuse. Il faut donc faire beaucoup d'itérations pour les méthodes itératives. A_1 et A_2 ne sont pas considérées car elles divergent, donc mettent un temps infini ... Le code n'a pas été mis dans le dossier pour ne laisser que les fichiers indispensables à la résolution, mais le code est disponible en figure 6.

On obtient alors, pour une précision fixée à 10^{-6}, les temps en secondes suivants :

A_i	Temps (Gauss)	Temps (Jacobi)	Temps (Gauss-Seidel)
A_3	0.000008	0.000147	0.000057
A_4	0.000001	0.000242	0.000355
A_6	0.000005	0.001066	0.000511

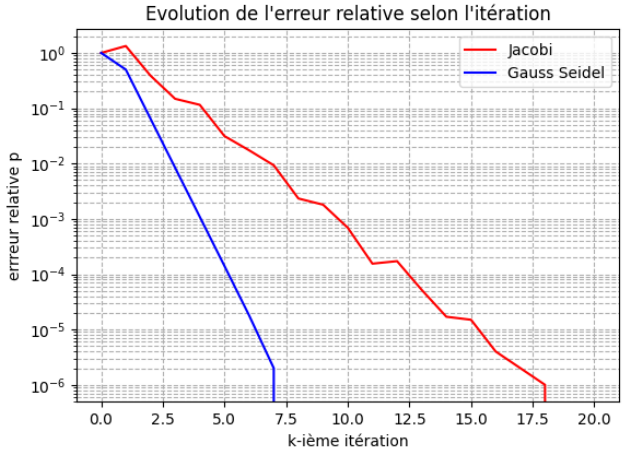
On voit que Gauss-Seidel est plus rapide que Jacobi en général, sauf pour A_4 comme ce qui était attendu. Mais le plus surprenant est que la méthode de Gauss est la plus rapide !

Ceci s'explique par le fait que les matrices testées ici sont de taille beaucoup trop petite pour que les méthodes itératives deviennent plus intéressantes à utiliser que les méthodes directes. Il suffirait alors de prendre des matrices de taille 10000 ou plus pour voir que Gauss est inutilisable. Ceci n'a pas été vérifié en pratique, mais en théorie, cela semble se vérifier du fait qu'en terme de complexité : $O(\text{Gauss}) > O(\text{Jacobi}) > O(\text{Gauss-Seidel})$. En effet, la complexité pour Gauss est beaucoup plus importante, comme énoncé au *TP* précédent.

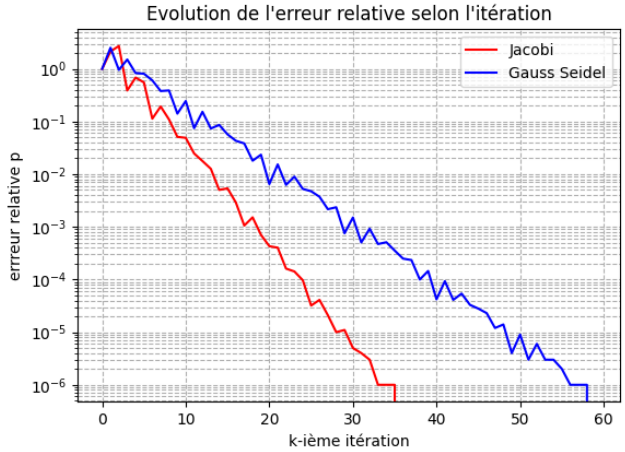


(a) A_1

(b) A_2

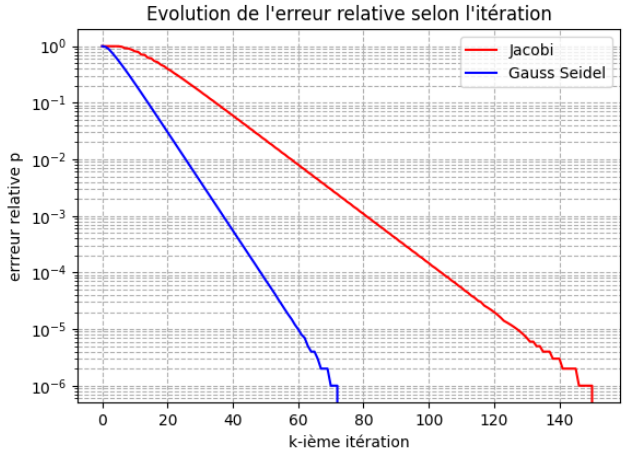


(c) A_3



(d) A_4

(e) A_5



(f) A_6

FIGURE 5 – Évolution des erreurs relatives selon la méthode itérative en échelle logarithmique

```c
#include <time.h>
#include "matriceTest.h"
#include"methodeJ_GS.h"
#include"gauss.h"

int main(){

    clock_t start, end;
    double elapsed;
    start = clock();     /* Lancement de la mesure */

    double **A3=matriceA3Dynamique();
    double**B3=matriceB3Dynamique();
    double **X3=methodeDeGauss(A3, TAILLE_MATRICE, B3, pow(10, -6));

    end = clock();    /* Arret de la mesure */
    elapsed = ((double)end - start) / CLOCKS_PER_SEC; /* Conversion en
       seconde*/
    printf("%f secondes entre le debut et la fin pour la methode \
    de Gauss avec A3.\n", elapsed);


    start = clock();

    double **A3p=matriceA3Dynamique();
    double**B3p=matriceB3Dynamique();
    double **X3p=methode_jacobi(A3p, B3p, TAILLE_MATRICE, pow(10, -6),
       1000);

    end = clock();
    elapsed = ((double)end - start) / CLOCKS_PER_SEC;
    printf("%f secondes entre le debut et la fin pour la methode \
    de Jacobi avec A3.\n", elapsed);


    start = clock();

    double **A3p2=matriceA3Dynamique();
    double**B3p2=matriceB3Dynamique();
    double **X3p2=methode_gauss_seidel(A3p2, B3p2, TAILLE_MATRICE, pow(10,
       -6), 1000);

    end = clock();
    elapsed = ((double)end - start) / CLOCKS_PER_SEC;
    printf("%f secondes entre le debut et la fin pour la methode \
    de Gauss-Seidel avec A3.\n", elapsed);

    /* Idem pour A4 et A6 + liberation des matrice */

    return EXIT_SUCCESS;
}
```

FIGURE 6 – CODE DU FICHIER `temps.c` POUR CALCULER LE TEMPS D'EXÉCUTION DES DIFFÉRENTES MÉTHODES

5 Conclusion générale

En conclusion, les différentes méthodes (directes et itératives) permettent de résoudre un système d'équations linéaires. Les méthodes directes ont pour avantage de pouvoir trouver la solution à coup sûr (si elle existe), mais ne garantissent pas de la trouver en un temps raisonnable. Les méthodes itératives, quant à elles, terminent en moins de temps, mais avec un résultat moins précis (même si on peut contrôler cette précision). Parfois, il se peut que les méthodes itératives ne trouvent pas la solution, même si elle existe. Dans ce *TP*, Gauss est plus rapide, car on a des matrices de petites tailles. Toutes les méthodes itératives ne se valent pas. En effet, on a pu constater que Jacobi est moins rapide que Gauss-Seidel dans la plupart des cas.

Il reste donc à l'utilisateur de choisir la méthode qu'il souhaite pour la résolution. Nous recommandons d'utiliser la méthode de Gauss pour les petites matrices. Néanmoins, pour les grandes matrices, une méthode itérative (de préférence celle de Gauss-Seidel) est plus adaptée, mais si on se rend compte que la méthode diverge, alors il faut se ramener à une méthode directe qui peut mettre plus ou moins longtemps selon la taille de la matrice.