Algorithmes numériques

Résolution de systèmes linéaires : Méthodes itératives

Dans ce dossier, nous allons vous présenter une implémentation en langage C des méthodes de Jacobi et de Gauss-Seidel, méthodes itératives pour la résolution de systèmes linéaires. Nous étudierons également leurs complexité, leurs stabilité, leurs vitesse de convergence et leur utilisation dans des cas pratiques judicieusement choisis. Nous conclurons par un face à face de ces deux méthodes pour déterminer laquelle privilégier en fonction de leurs vitesse de convergence.

Dossier et codes réalisés par :

BAYLE Nathanaël
CORCOS Ludovic

L2 Informatique
Université Clermont-Auvergne

Table des matières

I - Introduction **2**

II - Rappel des méthodes **2**

Matrice à diagonale strictement dominante 2

Méthode de Jacobi 3

Méthode de Gauss-Seidel 6

Conditions de convergence et d'arrêt 8

III - Présentation de nos programmes et algorithmes **9**

Fichier main.c : 9

Fichier matrices.c : 9

Fichier affichage.c : 10

Fichier jacobi.c 10

Fichier gauss-seidel.c : 10

IV - Mise en route sur des matrices pertinentes **11**

Méthode de Jacobi = 0,0001 11

Méthode de Gauss - Seidel = 0,0001 12

Etude de la convergence selon le paramètre 13

V - Analyse des résultats **13**

VI - Conclusion (méthodes itératives) **15**

VII - Conclusion (méthodes directes versus méthodes indirectes) **16**

I - Introduction

Il arrive souvent lors de calcul que nous soyons obligés de donner une valeur approchée de la solution (0,333333 pour 1/3 par exemple), c'est ce que l'on appelle la représentation des nombres en virgule flottante (8 ou 16 chiffres significatifs après la virgule). Il se peut également qu'une méthode employée pour résoudre un problème ne nous amène pas exactement à la solution mais à pas loin d'elle... On dit alors que l'on a une approximation de la solution.

Pour les systèmes linéaires par exemple nous nous intéresserons à deux types de méthodes bien distinctes :

- **Les méthodes directes :** qui consistent en des algorithmes qui permettent de calculer la solution exacte (mis à part les erreurs d'arrondis en virgule flottante) en un nombre fini d'opérations, ce que l'on a étudié dans le dossier précédent.

- **Les méthodes itératives :** qui consistent en l'approche de la solution exacte x par une suite de $(x_n)_{n \in \mathbb{N}}$ solutions approchées. Nous arrêtons alors les calculs lorsque nous atteignons un certain x_{n0} . Nous avons alors une erreur de troncature due à l'approximation de la solution exacte que nous mesurons à l'aide d'une norme appropriée (suivant la dimension de l'espace dans lequel x se trouve) $\| x - x_{n0} \|$. C'est justement à ces méthodes que l'on va consacrer ce dossier, les méthodes de Jacobi et de Gauss-Seidel dans la résolution de système linéaire du type $Ax = b$.

II - Rappel des méthodes

a) Matrice à diagonale strictement dominante

Avant de vous expliquer le fonctionnement des méthodes de Jacobi et de Gauss-Seidel, nous allons voir comment est définie une matrice à diagonale strictement dominante. En effet, c'est un critère qui va nous permettre de vérifier l'exactitude des méthodes décrites ci-dessous.

Le lemme « d'Hadamard » nous dit ceci :
" Si $A = \left((a_{i,j})_{i,j \in [1,n]} \right)$ est une matrice à diagonale strictement dominante alors A est inversible."

En d'autres termes, un matrice A est dite à diagonale strictement dominante si :

$$\forall i, \ 1 \leq i \leq n, \ |a_{ii}| > \sum_{j=1, j \neq i}^{n} |a_{ij}|$$

Voyons sur un exemple concret, soit la matrice A suivante :

$$A = \begin{pmatrix} -4 & 2 & 1 \\ 1 & 6 & 2 \\ 1 & -2 & 5 \end{pmatrix} \qquad \begin{aligned} |-4| &> |2| + |1| \\ |6| &> |1| + |2| \\ |5| &> |1| + |-2| \end{aligned}$$

La matrice A est à diagonale strictement dominante, comme le prouve la supériorité des valeurs absolues de la diagonale par rapport à la somme des valeurs absolues de la ligne correspondante.

b) Méthode de Jacobi

Soit (x_1, x_2, x_3) la solution du système ci-dessous. Si on connaît deux coordonnées de la solution, il est possible de calculer la troisième.

$$\begin{pmatrix} 1 & 1 & 0 \\ -1 & -1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix}$$

La ligne 1 donne : $x_1 = \dfrac{0 - 1 \cdot x_2 - 0 \cdot x_3}{1}$

La ligne 2 donne : $x_2 = \dfrac{1 + 1 \cdot x_1 - 1 \cdot x_3}{-1}$

La ligne 3 donne : $x_3 = \dfrac{2 - 0 \cdot x_1 - 1 \cdot x_2}{1}$

Si on ne connaît pas de valeurs de la solution, mais que l'on en a des valeurs approchées, il est possible d'itérer, c'est à dire choisir une nouvelle approximation (x_1, x_2, x_3) plus proche du résultat en réutilisant les formules ci-dessus.
La méthode de Jacobi consiste à effectuer des itérations successives tant que l'on est pas assez proche du résultat.

On remarque que si les éléments diagonaux de A sont non nuls, le système linéaire $Ax = b$ est équivalent à :

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1,\, j \neq i}^{n} a_{ij} x_j \right), \qquad i = 1, \ldots, n.$$

Pour une donnée initiale $x^{(0)}$ choisie, on calcule $x^{(k+1)}$ par :

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1,\, j \neq i}^{n} a_{ij} x_j^{(k)} \right), \qquad i = 1, \ldots, n.$$

C'est ce qui nous permet de définir la décomposition suivante pour A :

- D la matrice contenant les éléments diagonaux de A
- E la matrice triangulaire inférieur contenant les éléments sous diagonaux de A
- F la matrice triangulaire supérieur contenant les éléments sur diagonaux de A

C'est-à-dire :

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & a_{ij} & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \qquad D = \begin{pmatrix} a_{11} & & 0 \\ & \ddots & \\ 0 & & a_{nn} \end{pmatrix}$$

$$E = \begin{pmatrix} 0 & & 0 \\ & \ddots & \\ a_{ij}\ (i>j) & & 0 \end{pmatrix} \qquad F = \begin{pmatrix} 0 & & a_{ij}\ (i<j) \\ & \ddots & \\ 0 & & 0 \end{pmatrix}$$

L'algorithme de Jacobi décompose la matrice en la somme $A = M - N$ et calcule la suite : $x^{k+1} = M^{-1} N x^k + M^{-1} b$

Avec : $\begin{cases} M = D \\ N = -E - F \end{cases}$

On appelle matrice de Jacobi la $J = D^{-1}(-E-F) = M^{-1}N$ matrice

Soit le système suivant à diagonale strictement dominante suivant : $\begin{pmatrix} 5 & -1 & 2 \\ 3 & 8 & -2 \\ 1 & 1 & 4 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ -25 \\ 6 \end{pmatrix}$

On en déduit les solutions suivantes :

$$x_1 = \frac{12 + x_2 - 2 \cdot x_3}{5}$$

$$x_2 = \frac{-25 - 3 \cdot x_1 + 2 \cdot x_3}{8}$$

$$x_3 = \frac{6 - x_1 - x_2}{4}$$

On peut dresser un tableau général de trois itérations successives (même si d'autres sont encore possible) en prenant comme condition initiale pour mieux visualiser le fonctionnement :

	1	2	3
x_1	0	$x_1 = \frac{12 + 0 - 2 \cdot 0}{5} = \frac{12}{5} = 2,4$	$x_1 = \frac{12 + (-3,125) - 2 \cdot (1,5)}{5} = 0,815$
x_2	0	$x_2 = -\frac{25}{8} = -3,125$	$x_2 = \frac{-25 - 3 \cdot (2,4) + 2 \cdot (1,5)}{8} = -3,65$
x_3	0	$x_3 = \frac{6}{4} = 1,5$	$x_3 = \frac{6 - 2,4 - (-3,125)}{4} = 1,68125$

Sachant que le vecteur solution (exacte via la méthode directe de Gauss) est $x_1 = 1, x_2 = -3, x_3 = 2$, on constate que plus on fait des itérations, plus on va se rapprocher de ce vecteur.

On peut représenter l'algorithme de Jacobi en pseudo-code :

Données : A, b, x^0, n, ε *et* $MAXITER$
Pour $i = 1$ à n faire
    $x_i^{nouv} := x_i^0$
it := 0
Tant que ($\| Ax^{nouv} - b \| > \varepsilon$) et ($it < MAXITER$) faire
    it := it + 1
    Pour $i = 1$ à n faire
        $x_i^{vieux} := x_i^{nouv}$
    Pour $i = 1$ à n faire

$$x_i^{nouv} := \frac{b_i - \sum_{j=1,\, j \neq i} a_{ij} x_j^{vieux}}{a_{ii}}$$

Fin tant que
Fin

On remarque que la méthode de Jacobi nécessite n mémoires pour le vecteur x^{vieux} et n mémoires pour le vecteur x^{nouv}.

c) Méthode de Gauss-Seidel

Partons de la méthode de Jacobi, le calcul des vecteurs $x^{(1)}$, $x^{(2)}$, ..., $x^{(n)}$ mène à la convergence, cela veut dire que chaque nouveau vecteur est meilleur que le précédent. On remarque dans la méthode de Jacobi que pour calculer la composante $x_2^{(2)}$ du vecteur $x^{(2)}$ on utilise celles de $x^{(1)}$ malgré le fait que $x_1^{(2)}$ est déjà été calculé et en plus, elle est meilleure que $x_1^{(1)}$. D'ici vient le principe de la méthode de Gauss-Seidel, on utilise chaque composante dès qu'elle sera calculée. Ainsi, pour calculer la composante $x_i^{(k+1)}$ on utilise toutes les composantes de $x_1^{(k+1)}$ à $x_{i-1}^{(k+1)}$ déjà calculé à l'itération $(k+1)$ en plus de celles $x_{i+1}^{(k)}$ à $x_n^{(k)}$ qui ne sont qu'à l'itération (k).

L'algorithme de Gauss-Seidel (GS) décompose la matrice en la somme $A = M - N$ et calcule la suite : $x^{k+1} = M^{-1}Nx^k + M^{-1}b$

Avec : $\begin{cases} M = D + E \\ N = -F \end{cases}$

On appelle matrice de Gauss-Seidel la matrice : $GS = (D+E)^{-1}(-F) = M^{-1}N$

On peut reprendre le système à diagonale strictement dominante vu précédemment dans la méthode Jacobi :

$$\begin{pmatrix} 5 & -1 & 2 \\ 3 & 8 & -2 \\ 1 & 1 & 4 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ -25 \\ 6 \end{pmatrix}$$

On en déduit les solutions suivantes :

$$x_1 = \frac{12 + x_2 - 2 \cdot x_3}{5}$$

$$x_2 = \frac{-25 - 3 \cdot x_1 + 2 \cdot x_3}{8}$$

$$x_3 = \frac{6 - x_1 - x_2}{4}$$

On peut dresser un tableau général de trois itérations successives (même si d'autres sont encore possible) en prenant comme condition initiale pour mieux visualiser le fonctionnement :

	1	2	3
x_1	0	$x_1 = \frac{12 + 0 - 2\cdot 0}{5} = \frac{12}{5} = 2,4$	$x_1 = \frac{12 + (-4.025) - 2\cdot(1,90625)}{5} = 0,8325$
x_2	0	$x_2 = \frac{-25 - 3\cdot(2,4) + 2\cdot(0)}{8} = -4,025$	$x_2 = \frac{-25 - 3\cdot(0,8325) + 2\cdot(1,90625)}{8} = -2,960625$
x_3	0	$x_3 = \frac{6 - 2,4 - (-4,025)}{4} = 1,90625$	$x_3 = 2,03203125$

Sachant que le vecteur solution (exacte via la méthode directe de Gauss) est $x_1 = 1, x_2 = -3, x_3 = 2$, on constate que plus on fait des itérations, plus on va se rapprocher de ce vecteur, mais cette fois ci, plus rapidement qu'avec la méthode Jacobi. En effet, à la 4ème itération, $x_1 = 0,9950\ldots$. On est très proche de 1. Cette méthode doit apparemment fonctionner plus rapidement.

On peut représenter l'algorithme de Gauss - Seidel en pseudo-code :

Données : $A,\ b,\ x^0,\ n,\ \varepsilon$ *et* $MAXITER$
Pour $i = 1$ à n faire
    $x_i^{nouv} := x_i^0$
it := 0
Tant que ($\| Ax^{nouv} - b \| > \varepsilon$) et ($it < MAXITER$) faire
    it := it + 1
    Pour $i = 1$ à n faire
        $x_i^{vieux} := x_i^{nouv}$
    Pour $i = 1$ à n faire

$$x_i^{nouv} := \frac{b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{nouv} - \sum_{j=j+1}^{n} a_{ij} x_j^{vieux}}{a_{ii}}$$

Fin tant que
Fin

Mais, cette fois-ci, on remarque que la méthode de Gauss-Seidel nécessite seulement n mémoires, la composante x_i^{nouv} prend la place de x_i^{vieux} qui ne sera pas utilisé pour le calcul de $x_{i+1}^{k+1}, x_{i+2}^{k+1}, \ldots, x_n^{k+1}$

d) Conditions de convergence et d'arrêt

Pour que les méthodes itératives de résolution des systèmes d'équations linéaires convergent quelque que soit le vecteur initial x^0 il faut que la matrice A soit diagonalement dominante ce qui est très facile à vérifier, comme nous l'avons vu au a).
On peut aussi dire que si A est une matrice symétrique définie positive alors la méthode de Gauss-Seidel converge quelque soit le vecteur initial x^0. Cependant, nous ne nous occuperons pas de ce cas, car l'intérêt est de pouvoir comparer les méthodes avec les mêmes conditions initiales.

Lorsqu'on programme un algorithme itératif il est vivement conseillé de prévoir au moins deux critères d'arrêt :

- Le premier critère consistera à arrêter les calculs si deux approximations successives de la solution sont « suffisamment proches » l'une de l'autre. On arrête les calculs pour ces méthodes lorsque la différence absolue entre deux itérations successives soit inférieure à une certaine précision ε donnée.

$$| x^{(n+1)} - x^{(n)} | < \varepsilon$$

- Le deuxième critère doit imposer que le programme soit arrêté après un nombre limité d'itérations. En effet, si la méthode produit une suite de vecteurs qui diverge, le programme s'arrêtera en général suite à un « overflow ». Cependant, le programme peut « cycler » : il n'y a ni convergence numérique ni « overflow ». La suite des approximations reproduit périodiquement une même succession de vecteurs ou donne naissance à une suite chaotique de vecteurs bornés. Dans de tels cas, si l'on ne limite pas le nombre d'itérations, le programme fonctionnera jusqu'à ce que l'ordinateur ait un problème physique. Ici le nombre maximum d'itérations est fixé à 500 000.

III - Présentation de nos programmes et algorithmes

- **Fichier main.c :**

Dans ce fichier, nous allons retrouver l'inclusion des bibliothèques et des autres fichiers qui sont nécessaires au bon fonctionnement du programme. Ligne 40 et 41, nous avons **l'allocation dynamique** des matrices colonnes x et B, nous verrons l'allocation de la matrice A juste après. Nous initialisons la variable **"time"** à la ligne 57, c'est grâce à elle et plus particulièrement aux variables start et end (cf. ligne 58) que nous allons pouvoir savoir combien de temps notre programme met pour tourner ! C'est ici que le choix de la méthode utilisée est effectué.

- **Fichier matrices.c :**

Ce fichier permet de générer les matrices de test du TP, mais également une matrice creuse avec au moins 70 % de valeurs nulles, et bien sûr la possibilité de pouvoir rentrer une matrice de taille $n \times n$ manuellement.
Pour les génération de la matrice creuse, on se sert de la fonction **"rand()"** qui fait partie de la bibliothèque **stdlib.h**. Voyons comment cela fonctionne :
On génère un nombre aléatoire entre 0 et 100, et si ce nombre est inférieur à 70 alors la fonction **alea()** renverra un 0. Si le nombre aléatoire généré est supérieur à 70, alors la fonction **alea()** renverra un nombre aléatoire entre 0 et 100.

```
5  int alea() {
6      int result;
7      const int MAX = 100, MIN = 0;
8      int n = (rand() % (MAX - MIN + 1)) + MIN;
9      if (n < 70){
10         result = 0;
11     }
12     else{
13         result = (rand() % (MAX - MIN + 1)) + MIN;
14     }
15     return result;
16 }
```

Nous avons également dans ce fichier, entre la ligne 29 et 35, la déclaration de **"matrice"** qui nous permet d'avoir une allocation dynamique. Nous rappelons que l'allocation dynamique permet de **demander "manuellement de la mémoire"** grâce à la bibliothèque **stdlib.h**.
La fonction malloc demande au système la permission d'utiliser de la mémoire.
La fonction free permet de lui indiquer que l'on n'a plus besoin de la mémoire qu'on avait demandée.

- **Fichier affichage.c :**

Nous avons choisi d'adopter une pseudo interface graphique, même si elle ne reste que dans le shell, et c'est justement ce fichier qui permet de gérer ces différentes options.

La fonction **afficher_systeme(double **A, float *b)** nous permet d'afficher le système sous forme matricielle après avoir choisi un type de matrice prédéfini. Il en est de même pour la fonction **afficher_vecteur(float *A).**

De la ligne 58 à 122, correspond l'affichage du menu permettant de choisir parmis les différentes matrices de test proposés. et à partir de la ligne 128, nous avons la présentation du deuxième menu nous permettant de choisir entre une résolution du système, par Gauss ou Cholesky.

- **Fichier jacobi.c**

Ce fichier correspond à l'implémentation de l'algorithme de Jacobi. Détaillons-le un peu :

Le prototype de cette méthode est : **jabobi(matriceA, matriceB, matriceX, ε, maximum d'itérations).**

Cet algorithme commence par vérifier si la matrice est à diagonale strictement dominante, qui, comme on a pu le voir précédemment est une condition nécessaire et suffisante de convergence. Pour se faire, il utilise la **fonction vabJ** (qui retourne la valeur absolue de flottant pris en argument) et la **fonction diagstrictdom** qui permet de sortir du programme si la matrice n'est pas à diagonale strictement dominante.

On a ligne 36 le début du calcul des itérations, **while ((norme(A, x, b) > E) && (iter < maxiter))** nous dit de continuer tant que la norme matricielle est supérieure à la valeur ε (i.e : tant que l'on n'a pas une assez bonne précision sur les résultats) et que le nombre d'itérations doit être strictement inférieur au nombre maximum autorisé. La partie suivante, effectue les calculs indiqués en première partie du document.

```
while ((norme(A, x, b) > E) && (iter < maxiter))
ret
     for (i = 0; i < taille; i++){
          somme = 0;
          for (j = 0; j < taille; j++){
               if (j != i) {
                    somme += A[i][j] * x[j];
               }
          }
          y[i] = (b[i] - somme) / A[i][i];
     }
     for (i = 0; i < taille; i++){
          x[i] = y[i];
          //printf("x[%d] = %f\n", i, x[i]);
     }
     iter++;
```

- **Fichier gauss-seidel.c :**

Comme nous l'avons dit précédemment, l'algorithme de Gauss - Seidel reprend la méthode de Jacobi. Le prototype de cette méthode est : **gauss_seidel(matriceA, matriceB, matriceX, ε, maximum d'itérations).**

Comme dans le fichier jacobi.c, cet algorithme commence par vérifier si la matrice est à

```
for (i = 0; i < taille; i++)
{
     x[i] = y[i];
}
for (i = 0; i < taille; i++)
{
     somme1 = 0;
     somme2 = 0;
     for (j = 0; j < i; j++)
     {
          somme1 += A[i][j] * y[j];
     }
     for (j = i + 1; j < taille; j++)
     {
          somme2 += A[i][j] * x[j];
     }
     y[i] = (b[i] - somme1 - somme2) / A[i][i];
}
```

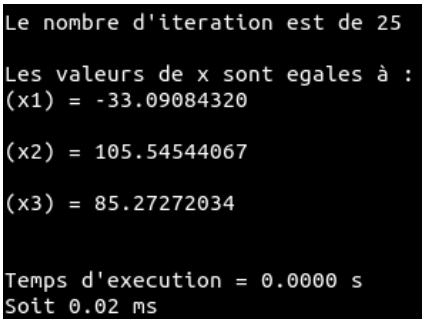
diagonale strictement dominante, il utilise donc la **fonction vabGS** (qui retourne la valeur absolue de flottant pris en argument) et la **fonction diagstrictdom** qui permet de sortir du programme si la matrice n'est pas à diagonale strictement dominante.
La modification remarquable est qu'ici, les valeurs déjà calculées initialement dans x[i] (qui sont mises dans la matrice temporaire y) sont automatiquement réinjectées dans la suite de l'algorithme pour le calcul de l'itération suivante.

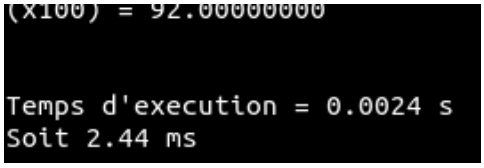
IV - Mise en route sur des matrices pertinentes

Nous allons à présent mettre en route nos algorithmes sur différentes matrices. Testons notre programme sur la matrice Bord Carrée vu que c'est la seule matrice du jeu d'essai qui est à diagonale strictement dominante, elle va nous permettre de comparer les deux méthodes implémentées.

- **Méthode de Jacobi** $\varepsilon = 0,0001$

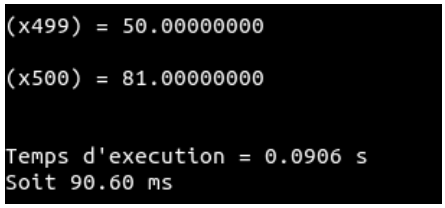
Pour la matrice Bord carrée de dimension **n = 3** avec remplissage aléatoire du vecteur B on a :
25 itérations en 0,02 ms.

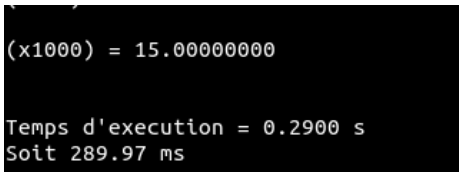




Pour la matrice Bord carrée de dimension **n = 100** avec remplissage aléatoire du vecteur B on a :
25 itérations en 2,44 ms

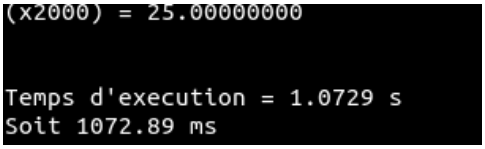
Pour la matrice Bord carrée de dimension **n = 500** avec remplissage aléatoire du vecteur B on a :
27 itérations en 90,6 ms.



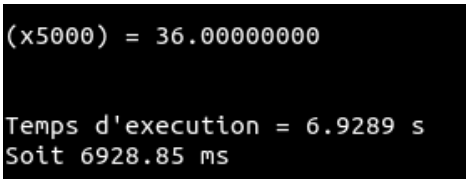


Pour la matrice Bord carrée de dimension **n = 1000** avec remplissage aléatoire du vecteur B on a :
26 itérations en 289,97 ms

Pour la matrice Bord carrée de dimension **n = 2000** avec remplissage aléatoire du vecteur B on a :
26 itérations en 1072,89 ms.

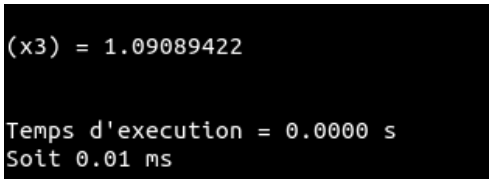


Pour la matrice Bord carrée de dimension **n = 5000** avec remplissage aléatoire du vecteur B on a :
26 itérations en 6928,85 ms.



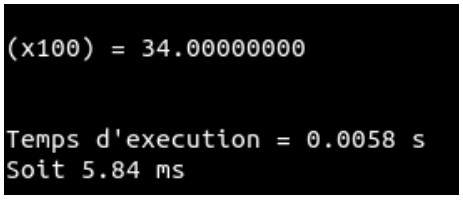
- **Méthode de Gauss - Seidel** $\varepsilon = 0,0001$

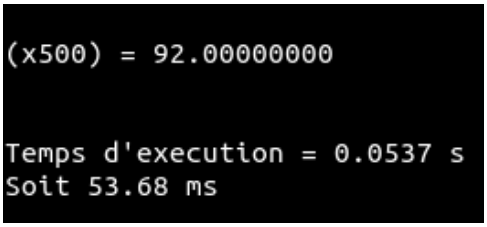
Nous faisons de même pour la méthode de Gauss - Seidel, nous allons tester l'algorithme sur la même matrice avec les mêmes valeurs de n que précédemment, comme ça nous allons pouvoir les comparer.



Pour la matrice Bord carrée de dimension **n = 3** avec remplissage aléatoire du vecteur B on a :
13 itérations en 0,01 ms.

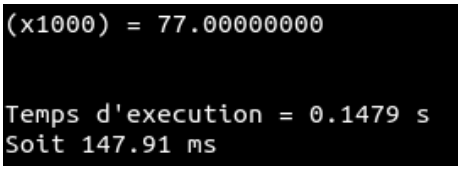
Pour la matrice Bord carrée de dimension **n = 100** avec remplissage aléatoire du vecteur B on a :
14 itérations en 5,84 ms.

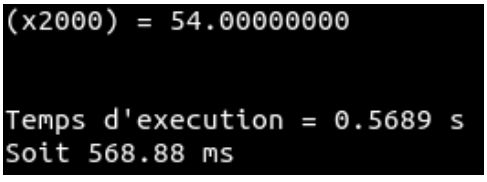




Pour la matrice Bord carrée de dimension **n = 500** avec remplissage aléatoire du vecteur B on a :
15 itérations en 53,68 ms/

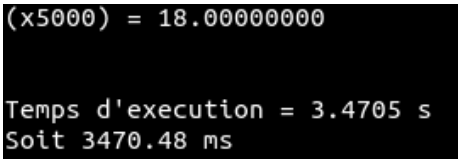
Pour la matrice Bord carrée de dimension **n = 1000** avec remplissage aléatoire du vecteur B on a :
14 itérations en 147,91 ms.





Pour la matrice Bord carrée de dimension **n = 2000** avec remplissage aléatoire du vecteur B on a :
14 itérations en 568,88 ms.

Pour la matrice Bord carrée de dimension **n = 5000** avec remplissage aléatoire du vecteur B on a :
14 itérations en 3470,48 ms.



- **Etude de la convergence selon le paramètre ε**

Pour pouvoir vérifier comment évolue la convergence des algorithmes, il nous suffit de modifier le paramètre ε dans les algorithmes de Jacobi et de Gauss - Seidel. Nous allons nous servir de la matrice Bord Carrée pour n = 15.
On peut dresser le tableau suivant :

	Nombre d'itérations Jacobi	Nombre d'itérations Gauss - Seidel
$\varepsilon = 0.1$	13	8
$\varepsilon = 0.01$	16	8
$\varepsilon = 0.001$	20	12
$\varepsilon = 0.0001$	27	14
$\varepsilon = 0.00001$	30	18
$\varepsilon = 0.000001$	32	16
$\varepsilon = 0.0000001$	33	18
$\varepsilon = 0.000000000000001$	#Erreur	#Erreur

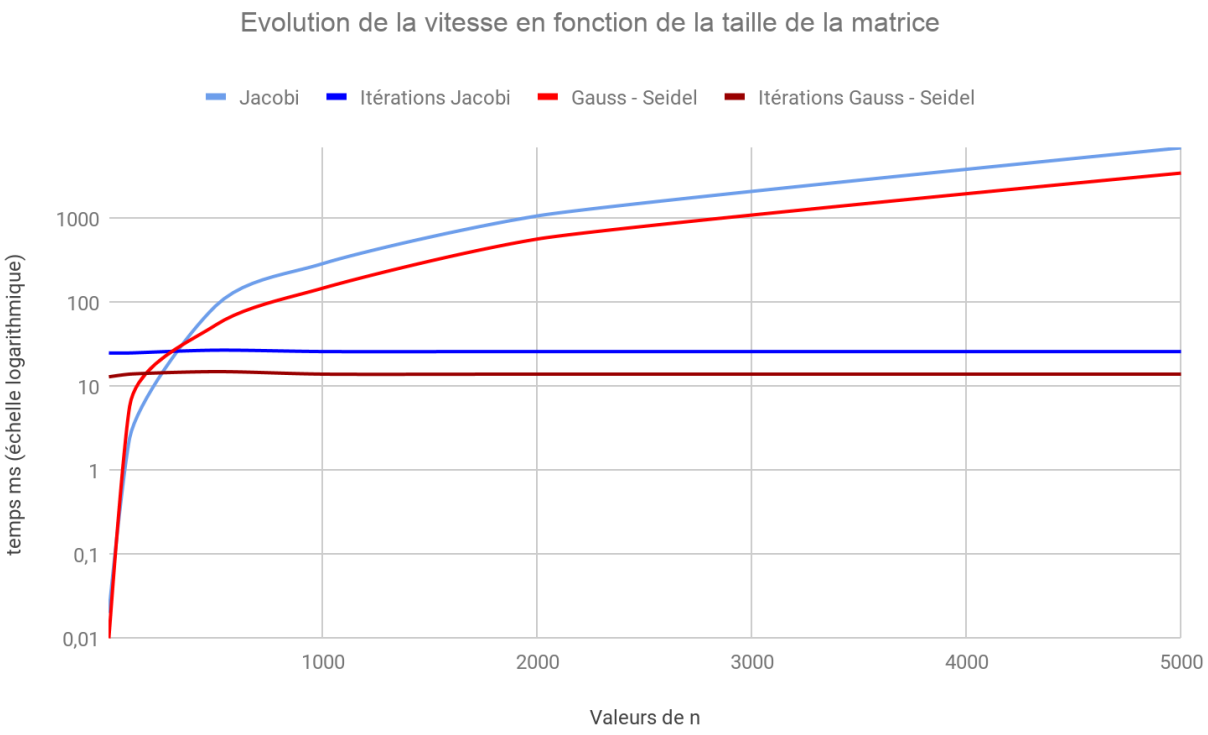
Signification de #Erreur : l'ordinateur sur lequel j'effectue les tests n'est pas assez puissant pour produire une précision $\varepsilon = 0.000000000000001$ sans avoir de BSOD.

V - Analyse des résultats

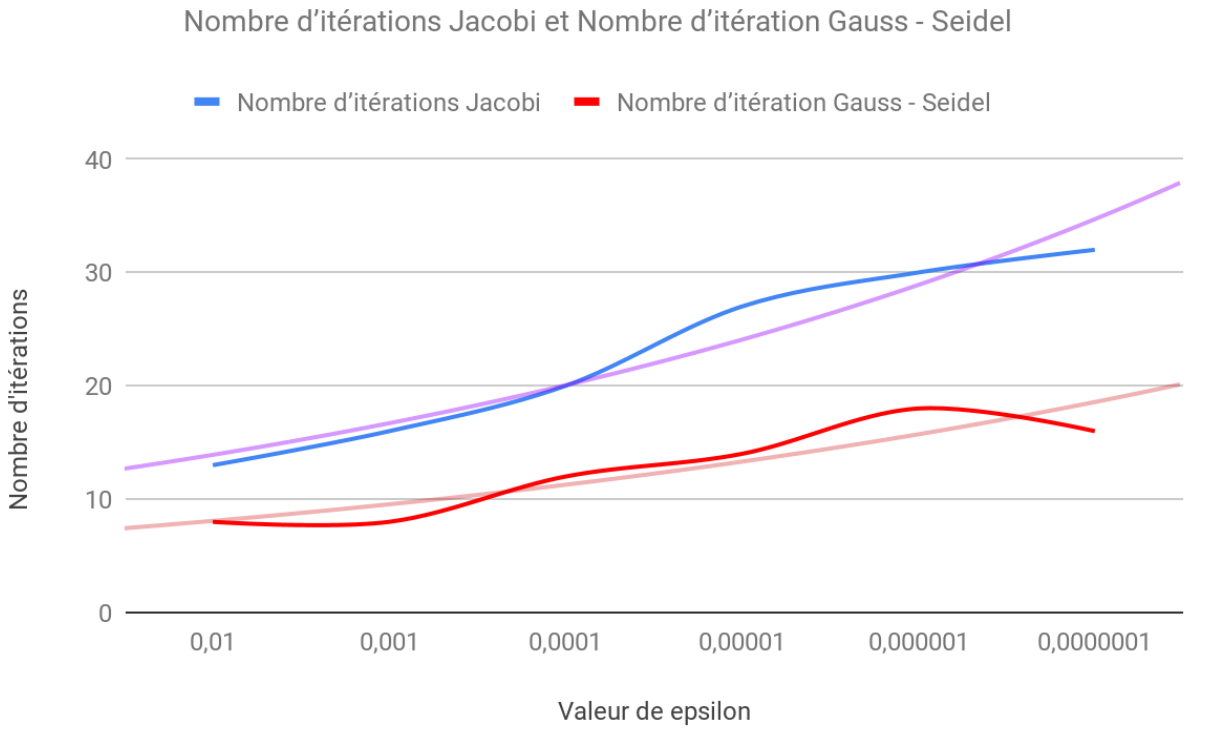
Comme nous avons pu le voir précédemment, nous avons les temps d'exécution des deux algorithmes implémentés, en fonction de matrices de tests générées ainsi qu'un tableau montrant le rapport en un ε petit et le nombre d'itérations nécessaires.
Nous réalisons donc **deux graphiques** :

Le premier nous montre une comparaison de la vitesse des deux algorithmes en fonction de la taille de la matrice et du nombre d'itérations. Nous utilisons une **échelle logarithmique** sur l'axe des ordonnées pour pouvoir représenter de manière simple les données ayant un grand écart.

Le deuxième nous montre l'évolution du nombre d'itérations en fonction de la valeur de précision ε avec les courbes de tendances associées.



On constate que plus la matrice est grande, plus le temps de calcul sera important, par ailleurs, même si le temps de calcul augmente, ce n'est pas le cas du nombre d'itérations pour produire une solution, en effet, il reste stable au fur et à mesure des tests, et donc ne dépend pas de la taille de la matrice.
On constate également que la méthode de Jacobi (courbes en bleu) est plus lente et demande plus d'itérations que la méthode de Gauss - Seidel.



Ce graphique nous aide à comprendre comment évolue le nombre le nombre d'itérations selon la méthode (Jacobi ou Gauss - Seidel), et on observe dans un premier temps que la méthode de Jacobi itère plus que la méthode de Gauss - Seidel (on confirme ici la lenteur de calcul), et dans un second temps que ce nombre d'itérations tend à évoluer de manière exponentielle comme nous l'indique les courbes de tendances (courbes moins opaques).

Les méthodes de Jacobi et de Gauss - Seidel ont un coût de l'ordre de $O(n^2)$ comme on peut le calculer au niveau des algorithmes écrits en pseudo-code.

Par ailleurs, vu que nous faisons du calcul numérique, il est impossible d'échapper aux erreurs de représentation des décimaux en langage machine.
Les deux sources d'erreur qui interviennent systématiquement dans le calcul numérique sont :

- **Les erreurs de troncature ou de discrétisation** qui proviennent de simplifications du modèle mathématique comme par exemple le remplacement d'une dérivée par une différence finie, le développement en série de Taylor limité, etc.
- **Les erreurs d'arrondi** qui proviennent du fait qu'il n'est pas possible de représenter (tous) les réels exactement dans un ordinateur

Afin de pouvoir effectuer des analyses d'erreurs d'arrondi, on fait des hypothèses sur l'exactitude des opérations arithmétiques de base. On suivra le modèle standard :

Type	Taille	Signe	Mantisse	Exposant	Unité d'arrondi u	Bornes
Simple	32 bits	1 bit	23 bits	8 bits	$2^{-24} \simeq 5.96 \times 10^{-8}$	$10^{\pm 38}$
Double	64 bits	1 bit	52 bits	11 bits	$2^{-53} \simeq 1.11 \times 10^{-16}$	$10^{\pm 308}$

Cependant, vu que dans le déroulement des méthodes itératives, on se base sur un vecteur initiale qui permet de converger, si une erreur de calcul apparaît, alors c'est comme si on recommençait d'un nouveau vecteur initial. En effet, l'erreur commise peut avoir comme conséquence que l'on redémarre le calcul avec un vecteur plus proche de la solution.

VI - Conclusion (méthodes itératives)

Le principal intérêt de ces méthodes est qu'elles sont stables en calcul et que les erreurs d'approximation décimales ne sont pas graves car elles ne débouchent pas sur des résultats totalement aberrants.
Cependant, les conditions de convergence sont difficiles à mettre en place, notamment l'utilisation de matrices à diagonale dominante qui est très restrictive, et l'utilisation de matrices symétriques définies positives nécessitent une analyse mathématique préalable. En conséquence, les méthodes itératives sont principalement utilisées pour

des matrices de grandes tailles et largement creuses. Les calculs croissent dans ce cas linéairement avec n (la taille de la matrice).
Il faut s'assurer que la convergence est suffisamment rapide pour que le temps de calcul ne soit pas excessif par rapport à la recherche de la solution. Cette comparaison des différentes méthodes itératives met en évidence que l'algorithme de Gauss-Seidel apporte une amélioration par rapport à celui de Jacobi, en permettant l'utilisation des nouvelles valeurs des variables déjà calculées lors d'une itération, et donc d'une économie en mémoire.

VII - Conclusion (méthodes directes versus méthodes indirectes)

Comme nous avons pu le voir dans ces deux dossiers, il existe plusieurs méthodes pour résoudre les systèmes linéaires du type $Ax = b$, les méthodes directes et indirectes.
Parmis les méthodes directes, nous avons vu les méthodes de Gauss et de Cholesky :

- Le coût de la méthode de Gauss est de $\frac{2n^3}{3}$
- Le coût de la méthode de Cholesky est de $\frac{n^3}{3}$
- La comparaison de leurs coûts respectifs favorisent la méthode de Cholesky dans le cas de matrices symétriques définies positives
- Aussi, les deux méthodes permettent de calculer l'inverse de la matrice A qui peut être stocké pour résoudre d'autres systèmes linéaires avec différents seconds membres b

Nous avons vu dans ce dossier également les méthodes itératives de Jacobi et de Gauss - Seidel :

- Le coût général de ces méthodes et de l'ordre de n^2
- La convergence n'est définie que sur les matrices à diagonales strictement dominantes
- Elles ont la particularité d'être stables sur des matrices creuses de grande taille
- Elles sont auto - correctrices, si on commet une erreur de calcul au cours du processus, tout se passe comme si on recommençait le calcul à partir d'un nouveau vecteur initial $x^{(0)}$
- Il est possible de choisir un critère pour affiner la précision des résultats

Pour pouvoir obtenir un résultat précis avec un temps acceptable, il est préférable d'utiliser la méthode de Cholesky (méthode directe) car elle a un coût en mémoire plus faible que la méthode de Gauss. Cependant, pour les matrices de grande taille il est préférable d'utiliser la méthode de Gauss - Seidel (méthode itérative) car elle converge plus rapidement lorsque ε est petit, ce qui implique qu'elle consomme peu de mémoire.