Compte rendu : Système Linéaire à Grande Dimension

Ben Khalifa Emna, Honakoko Giovanni

09/01/2025

Table des matières

1 Introduction

1.1 Notations

Dans le rapport qui suit, on considèrera :

$$A := (a_{ij})_{1 \leq j \leq i \leq n} \in M_n(\mathbb{R})$$
$$x := (x_j)_{1 \leq j \leq n} \in \mathbb{R}^n$$
$$b := (b_j)_{1 \leq j \leq n} \in \mathbb{R}^n$$

Soient $D, L, U \in M_n(\mathbb{R})$, où :

- D est la **diagonale** de A.
- L est la partie **triangulaire inférieure** de A .
- U est la partie **triangulaire supérieure** de A.

Remarque

Dans la suite de ce rapport pour chacun des test de nos fonctions. Nous avons utilisé une matrice tridiagonale représentant le Laplacien de différence finie de second ordre, qui s'approxime en discrétisant l'intervalle de valeur considéré à l'aide d'une formule symétrique.

1.2 Objectif

Le but de cette semaine de projet était d'étudier les méthodes de résolution numérique pour résoudre des systèmes linéaires en grande dimension, du type :

$$Ax = b \tag{1}$$

Afin de résoudre ce type d'équation, on a étudié la méthode de :

- **Jacobi** *(dense)*
- **Jacobi** *(sparse)*
- **GS** ou **Gauss-Seidel**
- **Surrelaxation successive** ou **SOR** *(Successive Over-Relaxation)*

Ces méthodes reposent sur deux principes afin de résoudre l'équation [1] qui sont :

- Le Pré-conditionnement.
- Le Splitting.

Remarque

Un point important à souligner que notre matrice du système linéaire est non plus A mais :

$$A_h = \frac{1}{h^2} A$$

où $h = \dfrac{1}{n+1}$ est le pas de subdivision de l'intervalle $[\![0, n]\!]$.

Informatiquement il est moins couteux en calcul d'avoir :

$$\begin{aligned}\frac{1}{h^2}Ax &= b \\ Ax &= h^2 b \\ Ax &= \tilde{b}, \text{ avec } \tilde{b} = h^2 b.\end{aligned}$$

C'est cette dernière égalité que l'on a utilisé dans nos calculs pour l'élaboration des graphiques à la fin de ce polycopié.

2 Théorème et définitions

2.1 Matrice strictement diagonale dominante

Soit $A \in M_n(\mathbb{R})$.
On dit que A est **strictement diagonale dominante** si et seulement si :

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

2.2 Rayon spectral

Soit $A \in M_n(\mathbb{R})$.
On définit le rayon spectral de la matrice A que l'on note $\rho(A)$, tels que :

$$\forall i \in [\![1, n]\!], \quad \rho(A) := \max(|\lambda_i|)$$

, où les λ_i sont les valeurs propres de A.

2.3 Matrice creuse

Soit $S \in M_n(\mathbb{R})$.
En informatique on appelle **matrice creuse** ou **sparse matrix** *(en anglais)*, une matrice dont on stock uniquement les données des coefficients non nuls, tels que :

$$\forall i, j \in [\![1, n-1]\!], \quad \texttt{S} = (\texttt{row\_index}[i], \texttt{col\_index}[j]) \quad \texttt{data}[i]$$

où `col_index` (*resp.* `row_index`) est la liste qui répertorie l'indice de la colonne (*resp.* ligne) de l'élément non nul.
Quant à `data` contient la valeur du coefficient non nul repéré par `row_index` et `col_index`.

2.4 Pré-conditionnement

Considérons $C, L, U \in M_n(\mathbb{R})$, avec C la matrice de pré-conditionnement.
La matrice A se décompose telle que : $A = D + L + U$. Néanmoins il sera plus judicieux d'adopter la forme :

$$A = D - \tilde{L} - \tilde{U}$$

où : $\begin{cases} \tilde{L} = -L \\ \tilde{U} = -U \end{cases}$

Les variables étant muettes on pose : $\begin{cases} L = \tilde{L}. \\ U = \tilde{U}. \end{cases}$

On a :

$$A = D - L - U \tag{2}$$

Le but est de multiplier C^{-1} à gauche de part et d'autre de [1].
Puis de décomposer A de façon réfléchie, de sorte à ce que du côté gauche de l'équation, on obtienne en facteur de x l'identité plus un résidus.
Enfin on isole $I \times x$, ce qui nous permettra d'obtenir une formule d'itération afin de déterminer x.

Exemple 1 :

Prenons $C = D$.
Ainsi on a :

$$\begin{aligned} D^{-1}(D - L - U)\, x &= D^{-1}b \\ \big(I - D^{-1}(L + U)\big)\, x &= D^{-1}b \\ x &= D^{-1}b + \underbrace{D^{-1}(L + U)}_{T} \end{aligned}$$

Ce qui nous donne la formule d'itération :

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{i \neq j}^{n} a_{ij} x_j^{(k)} \right), \forall i \in [\![1, n]\!] \tag{3}$$

On appelle T matrice d'itération de système linéaire.

2.5 Splitting

Le **Splitting** se base uniquement sur une décomposition intelligente de A.
Illustrons cela d'un exemple.

Exemple 2 :

Soient $M, N \in M_n(\mathbb{R})$, tels que :

$$A = M - N \tag{4}$$

et :

$$M = D - L$$

D'où :

$$\begin{aligned} A &= D - L - N \\ D - L - U &= D - L - N \\ N &= U \end{aligned}$$

En substituant nos termes dans [1], on a :

$$\begin{aligned} Ax &= b \\ (M-N)x &= b \\ (D-L-U)x &= b \\ Dx &= b+L+U \\ x &= D^{-1}b + +D^{-1}(L+U)\,x \end{aligned}$$

Ce qui nous donne encore une fois la formule itérative [3].

3 Méthode de Jacobi (dense)

On l'appelle **dense**, car en Python on stockera A comme une matrice pleine *(dense)*. L'exemple 1 illustre parfaitement le déroulement de la méthode de Jacobi. Elle s'aide d'un pré-conditionneur et nous donne la formule itérative [3].

- Une **condition suffisante** pour que la méthode de Jacobi converge est que la matrice A de [1] soit **strictement diagonale dominante**.
- Une **condition nécessaire et suffisante** est que :

  $$\rho(T) < 1$$

  , où T est la matrice d'itération de la méthode de Jacobi (cf.exemple 1).

3.1 Implémentation n°1 : Jacobi dense

La fonction `jacobi_method` implémente la méthode de Jacobi.

Arguments :

- `A` : Matrice du système linéaire.
- `b` : Vecteur second membre.
- `x0` : Vecteur des conditions initiales.
- `tol` : Tolérance définissant le critère d'arrêt basé sur la précision.
- `max_iter` : Maximum d'itération.

Retours :

- `x` : Vecteur des solutions approximées.
- `iterations` : Nombre d'itérations effectuées.
- `errors` : Liste des erreurs entre la solution exacte et approchée pour chaque itéré.

Nous avons programmé la formule [3],en exprimant la somme des termes non diagonaux $\sum_{i\neq j}^{n} a_{ij}x_j^{(k)}$:

$$\sum_{i\neq j}^{n} a_{ij}x_j^{(k)} = S1 + S2, \begin{cases} S1 = \sum_{i\neq j}^{n} a_{ij}x_j^{(k)} \\ S2 = \sum_{i\neq j}^{n} a_{ij}x_j^{(k)} \end{cases} \tag{5}$$

Le code est composé de 3 boucles et à chaque itération :

- La nouvelle valeur de x_{new} est calculée à partir des termes diagonaux et des termes non diagonaux de A.
- L'erreur $\|x_{new} - x\|$ est calculée.
- Si l'erreur est inférieure à une certaine tolérance (tol), alors le la méthode s'arrête.

En realité la vérification `error < tol` permet de vérifier la convergence de la méthode, en deçà de la tolérance fixée la méthode diverge et on arrête le processus.

3.2 Implémentation n°2 : Jacobi dense avec produit scalaire

La fonction `jacobi_method_produit_scalaire` implémente également la méthode de Jacobi via l'aide d'un produit scalaire.

Arguments :

- `A` : Matrice du système linéaire.
- `b` : Vecteur second membre.
- `x0` : Vecteur des conditions initiales.
- `tol` : Tolérance définissant le critère d'arrêt basé sur la précision.
- `max_iter` : Maximum d'itération.

Retours :

- `x` : Vecteur des solutions approximées.
- `iterations` : Nombre d'itérations effectuées.
- `errors` : Liste des erreurs entre la solution exacte et approchée pour chaque itéré.

Cette implémentation fait sensiblement la même chose que la précédente excepté le fait que l'on calcul la somme des termes non diagonaux avec un produit scalaire. Pour tout $i \in [\![1, n]\!]$:

$$\begin{aligned} \sum_{i\neq j}^{n} a_{ij}x_j &= \sum_{j=1}^{i-1} a_{ij}x_j + \sum_{j=i+1}^{n} a_{ij}x_j \\ &= \sum_{1\leq i\leq j\leq n}^{n} a_{ij}x_j - a_{ii} \\ &= \langle A, x\rangle - a_{ii} \end{aligned}$$

C'est la dernière ligne de ce calcul que l'on code avec :

$$\forall i \in [\![1, n-1]\!], \quad \boxed{\texttt{sum\_ax = np.dot(A[i,:],x)-A[i,i]*x[i]}}$$

3.3 Différences entre les 2 implémentations

Pour comparer nos deux méthodes on a pris pour matrice d'entrée une matrice carrée tridiagonale telle que :

$$M_T = \begin{pmatrix} 5 & -1 & 0 & \dots & 0 \\ -1 & 10 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & -1 \\ 0 & \dots & 0 & -1 & 5n \end{pmatrix}$$

On remarquera que les termes diagonaux s'exprime comme :

$$\forall i \in [\![1, n-1]\!], \quad m_{ii} := 5(i+1)$$

Pour générer ce type de matrice nous avons créer une fonction `generate_linear_system(n)` à l'intérieur laquelle prenant en entrée la dimension de la matrice carrée dans le but de retourner M_T.

Par ailleurs `n`= 110 est la taille maximum possible pour M_T avant que l'on perde la convergence de la méthode.
Cela se voit graphiquement, de plus $\rho(T) \geq 1$ ce qui nous assure la divergence de la méthode.

Comparaison temporelle

- Temps d'exécution implémentation n°1 : $t \simeq 3,776$ s
- Temps d'exécution implémentation n°2 : $t \simeq 0,191$ s

La première implémentation met plus de temps à calculer que la deuxième étant donné la présence de 3 boucle **for**. Pourtant les deux ont la même complexité qui est de l'ordre de $O(n^2)$.

Remarque

1. **Convergence dépendante de** A : Le nombre d'itérations k dépend de la matrice A *(plus précisément de son rayon spectral)*. Une matrice mal conditionnée peut ralentir la convergence.
2. **Mémoire utilisée** : Le code stocke la matrice A ce qui a une complexité de $O(n^2)$ et les vecteurs `x`, `x_new`, `b` en $O(n)$. Pour la liste des erreurs on a du $O(k)$. La mémoire totale est donc dominée $O(n^2)$.
3. **Amélioration possible** : Si A est une matrice creuse *(sparse)*, la complexité peut être réduite en exploitant les zéros pour éviter de parcourir les entrées inutiles.

Ce qui nous amène à :

3.4 Implémentation n°3 : Jacobi Sparse

Il s'agit d'une transposition de la méthode de Jacobi pour les matrices denses appliquées aux matrices creuses.

Arguments :

- `A` : Matrice creuse du système linéaire.
- `b` : Vecteur second membre.
- `x0` : Vecteur des conditions initiales.
- `tol` : Tolérance définissant le critère d'arrêt basé sur la précision.
- `max_iter` : Maximum d'itération.

Retours :

- `x` : Vecteur des solutions approximées.
- `iterations` : Nombre d'itérations effectuées.
- `times taken` : Temps pris pour chaque itération.

Nous avons donc retranscris matriciellement en Python la formule formule [1] :

$$x^{(k+1)} = D^{-1}(A - D)x^{(k)} + D^{-1}b$$

En appliquant les opérations, les méthodes et les fonctions adéquates aux matrices creuses comme :

- `diags` : donne la diagonale d'une matrice creuse ou en génère une matrice diagonale de type `sparse matrix`
- @ : Permettant d'effectuer le produit matricielle *(au sens des Mathématiques)*.

Pour cette implémentation nous avons également créer une fonction `generate_corrected_sparse_tridiagonal_matrix` telle que :

Arguments :

- `n` : Dimension de la matrice.
- `diagonal_value` : Valeur diagonale.
- `off_diagonal_value` : Valeur non diagonale.

Retours :

- `As` : Matrice creuse tridiagonale.
- `A_dense` : Matrice dense tridiagonale.
- `b` : Vecteur second membre du système linéaire.

Ainsi avec cette fonction on a pu générer un matrice tridiagonale creuse ainsi que sa version dense pour comparer les fonctions `jacobi_method_produit_scalaire` et `jacobi_sparse`.

Comparaison Temporelle

- Temps d'exécution implémentation n°2 : $t \simeq 2,243$ s
- Temps d'exécution implémentation n°3 : $t \simeq 0,236$ s

4 Méthode de Gauss-Seidel

La méthode de Gauss-Seidel se base sur le pré-conditionneur :

$$C = D - L \text{ ou } C = D - U$$

En l'appliquant [1], il en résulte :

$$\begin{aligned}(D-L)^{-1}(D-U-L)\,x &= (D-L)^{-1}\,b \\ \left(I-(D-L)^{-1}\,U\right)x &= (D-L)^{-1}\,b \\ x &= (D-L)^{-1}\,b + \underbrace{(D-L)^{-1}}_{T}Ux\end{aligned}$$

Traduite sous forme itérative cela donne :

$$\forall i \in [\![1,n]\!],\ \ x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right) \tag{6}$$

On remarque que de la récursivité est introduite dans la formule itérative que l'on avait jusqu'à présent.

4.1 Implémentation n°4 : Gauss-Seidel (GS)

Arguments :

- `A` : Matrice creuse du système linéaire.
- `b` : Vecteur second membre.
- `x0` : Vecteur des conditions initiales.
- `tol` : Tolérance définissant le critère d'arrêt basé sur la précision.
- `max_iter` : Maximum d'itération.

Retours :

- `x` : Vecteur des solutions approximées.
- `k` : Nombre d'itérations effectuées.
- `times taken` : Temps pris pour chaque itération.
- `errors` : Liste des erreurs entre la solution exacte et approchée pour chaque itéré.

En se basant sur l'implémentation de la méthode de jacobi avec produit scalaire, on arrive à exprimer les deux sommes de [6]. Le calcul s'effectue dans une boucle `for` sur $i \in [\![1, n-1]\!]$:

```
s1 = A[i, :i].dot(x_new[:i]).item()
s2 = A[i, i+1:].dot(x_new[i+1:]).item()
```

A part ce morceau de code le reste est identique aux implémentations de la méthode de Jacobi. On voit que Gauss-Seidel s'exécute en moins d'itérations que la méthode de Jacobi sparse, néanmoins il reste plus lent.

4.2 Différences avec les autres méthodes

5 Méthode de Surrelaxation successive (SOR)

La méthode Surrelaxation successive se construit sur la base d'un pré-conditionneur ou d'un splitting et un paramètre de relaxation $\omega \in]0, 2[$.

5.0.1 SOR avec splitting

Pour simplifier les calculs de l'ordinateur on prendra :

$$C = (D - L\omega)$$

A la fin on aura la méthode SOR à un facteur ω près. On décompose A comme suit :

$$\begin{aligned} \omega A &= \omega M - \omega N \\ \omega(D - L - U) &= D - L\omega - N \\ D(\omega - 1) - \omega U &= -N \\ N &= \frac{1}{\omega} D(1 - \omega) + U \end{aligned}$$

Le système linéaire donne :

$$\begin{aligned} Ax &= b \\ (M + N)x &= b \\ Mx &= Nx + b \\ x &= M^{-1}Nx + M^{-1}b \end{aligned}$$

5.1 Implémentation de SOR

La fonction `SOR_sparse_with_error` implémente la méthode SOR.

Arguments :

- `A` : Matrice creuse du système linéaire.
- `b` : Vecteur second membre.
- `x0` : Vecteur des conditions initiales.
- `tol` : Tolérance définissant le critère d'arrêt basé sur la précision.
- `max_iter` : Maximum d'itération.

Retours :

- `x` : Vecteur des solutions approximées.
- `k` : Nombre d'itérations effectuées.
- `times taken` : Temps pris pour chaque itération.
- `errors` : Liste des erreurs entre la solution exacte et approchée pour chaque itéré.

La structure du code est strictement la même que pour Gauss-Seidel avec seulement une ligne de plus :

$$\begin{aligned} \texttt{s}[i] &= (\texttt{b}[i] - \texttt{s1} - \texttt{s2})/\texttt{A}[i,i] \\ \texttt{x\_new}[i] &= \texttt{omega} * \texttt{s}[i] + (1 - \texttt{omega}) * \texttt{x}[i] \end{aligned}$$

Ce qui est l'expression de l'itéré précédent, et notre nouvelle itéré est `x_new` dont la formule se retrouve également modifié du au paramètre de relaxation.

Selon le paramètre ω on a une convergence plus ou moins de la méthode SOR. Et il existe une formule pour calculer le ω optimal pour une matrice tridiagonal. On le notera $\omega_{\texttt{opti}}$ et on l'exprime tel :

$$\omega_{\texttt{opti}} = \frac{2}{1 + \sqrt{1 - \rho(T_J)^2}} > 1$$

avec T_J la matrice d'itération de la méthode de Jacobi.

6 Comparaison graphique

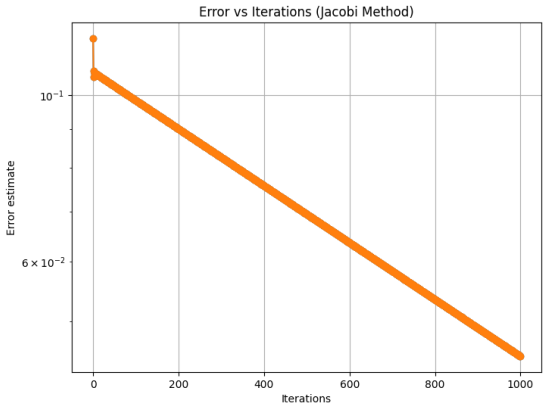

FIGURE 1 – Méthode de Jacobi dense pour matrice de dimension 110×110

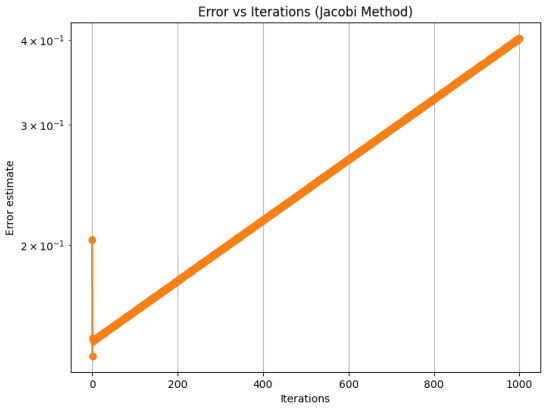

FIGURE 2 – Méthode de Jacobi dense pour matrice de dimension 111×111

Sur ces deux graphiques on a prit une matrice carrée tridiagonale de dimension $n = 110$, puis $n = 110$ dans le but de mettre en exergue la divergence de la méthode de Jacobi à partir d'une certaine taille de matrice.

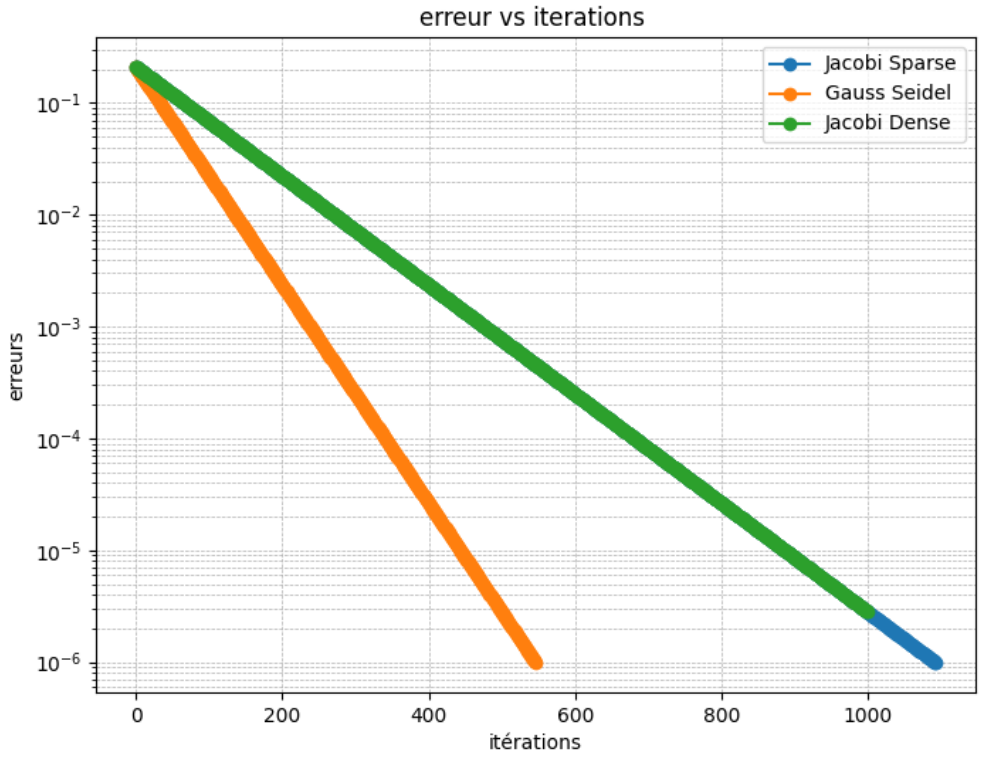

FIGURE 3 – Décroissance des erreurs en fonction du nombre d'itérations

Ce graphique corrobore bien le fait que la méthode de Gauss-Seidel s'opère en moins d'itération que la méthode de Jacobi dans sa version dense ou sparse.

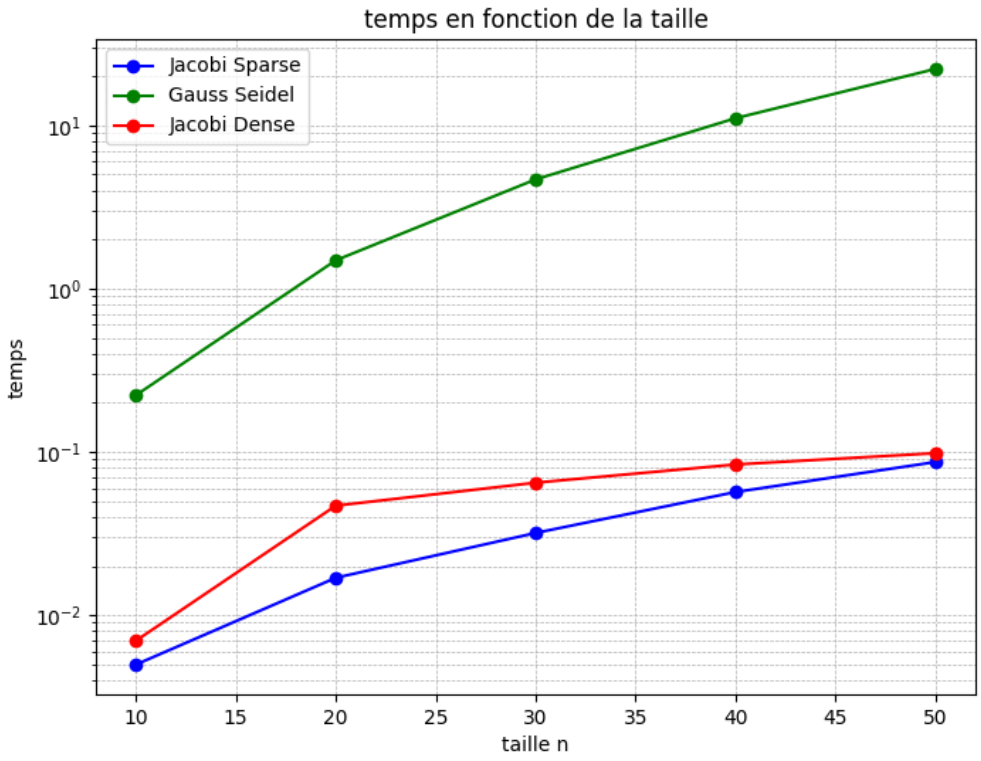

FIGURE 4 – Comparaison des méthode GS,JS,JD

Ici on voit bien que la méthode de Jacobi sparse est la plus rapide à opérer, et que Gauss-Seidel est la plus lente.

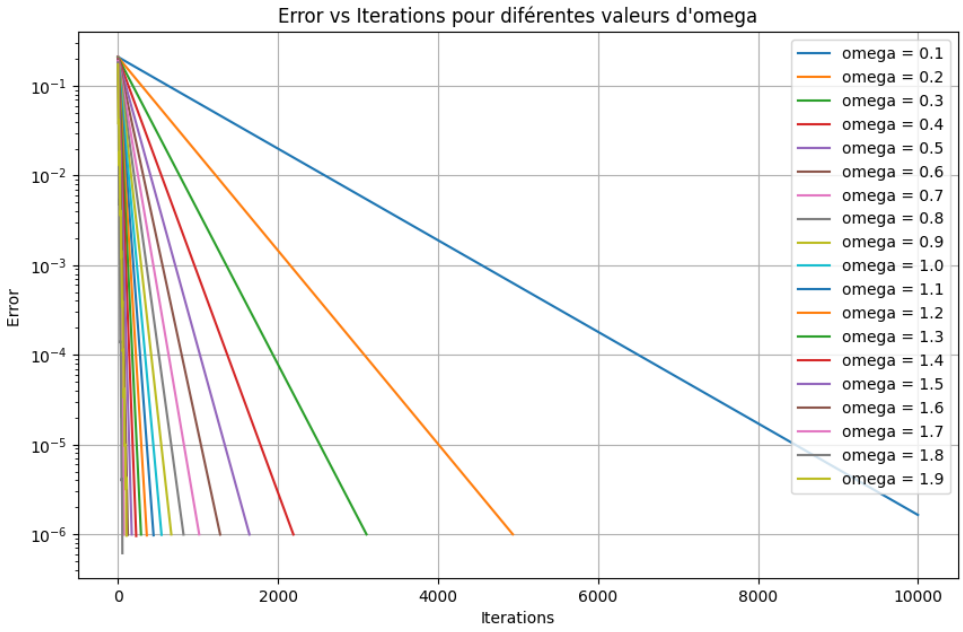

FIGURE 5 – Évolution de l'erreur en fonction du nombre d'itérations pour différentes ω

On remarque pour une matrice de dimension fixée, que la méthode SOR converge plus rapidement pour $\omega = 1,8$. On l'observe mieux sur le graphique suivant.

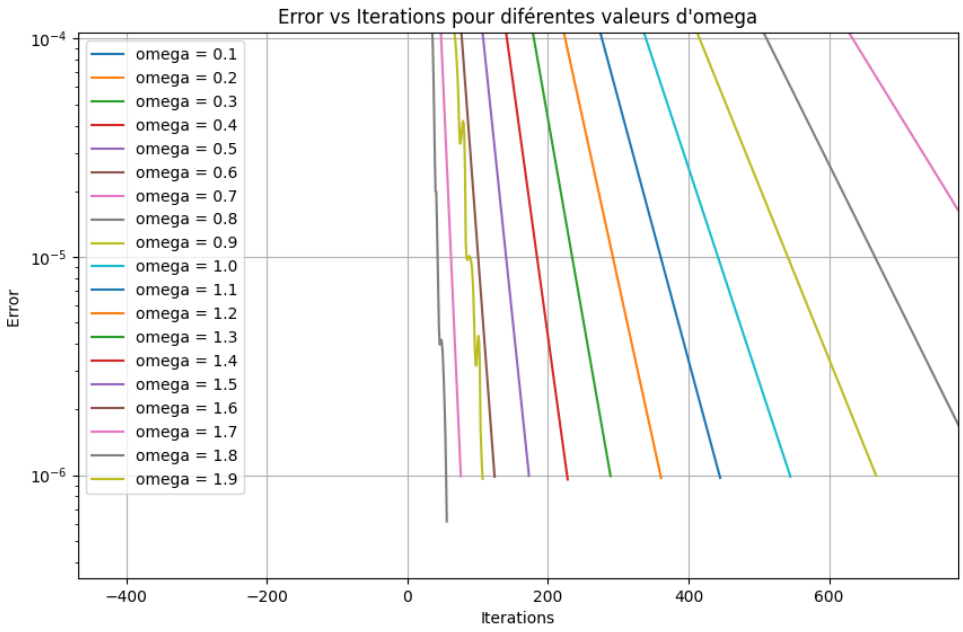

FIGURE 6 – Zoom sur l'évolution de l'erreur en fonction du nombre d'itérations pour différentes ω

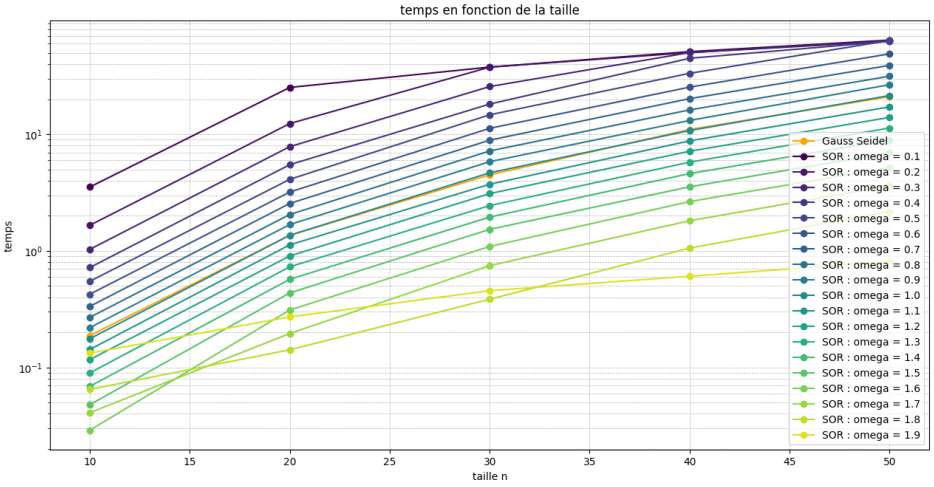

FIGURE 7 – Comparaison des méthodes de GS, SOR pour différents ω

Ici on observe plus généralement le comportement symétrique de la méthode SOR autour de $\omega = 1$.