Rapport du projet SLGD

COSTANTIN Perline, ZIAD Zineb
MAM 3

Introduction :

Dans le cadre de ce projet “Système linéaire en grande dimension” nous avons considéré le système linéaire $Ax = b$ pour des matrices A de grande dimension et étudié plusieurs méthodes afin de le résoudre, à savoir :

- I) Méthode de Jacobi sur des matrices denses
- II) Méthode de Jacobi sur des matrices sparses
- III) Méthode de Gauss-Seidel
- IV) Méthode SOR

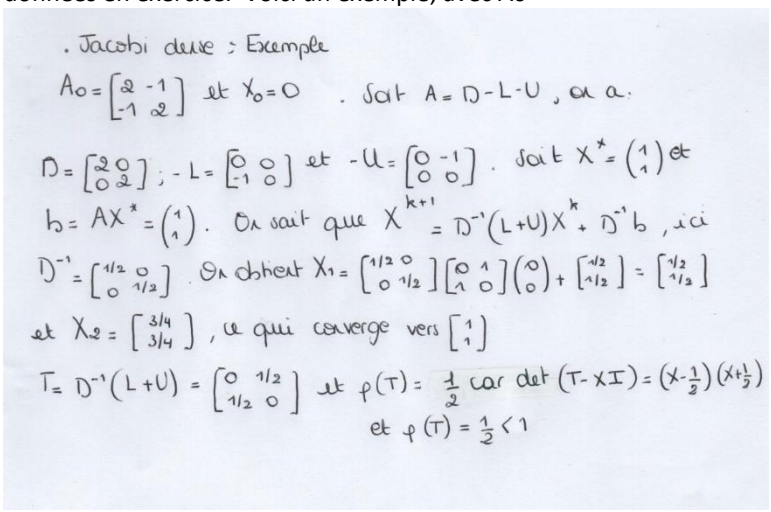
Nous avons ensuite implémenté ces méthodes en Python et analysé les résultats obtenus sur différents points, présentés dans ce rapport.

I) Méthode de Jacobi sur des matrices denses

1. Now you try n°1 :

Theoretical convergence of Jacobi: Now you try

Pour nous familiariser avec la méthode, nous avons fait à la main les quatre premières itérations des matrices données en exercice. Voici un exemple, avec A0



2. Now you try n°2 :

Jacobi Method Convergence Investigation: Now you try

Nous considérons le système linéaire $Ax = b$ où b est de dimension n et la matrice A à la structure suivante :

$a_{i,i}$ = 5*(i+1) , pour i=0,….,n-1 et $a_{i,j}$ = -1 pour i ≠ j

La dominance stricte est vérifiée si la valeur absolue de la somme de tous les éléments hors-diagonale d'une même ligne est inférieure à l'élément de la diagonale.
Dans le cas étudié, pour chaque ligne de A, les éléments hors-diagonale sont -1. Ainsi, la somme des valeurs absolues de ces éléments sur une ligne vaut n-1 (n-1 éléments égaux à -1), donc la condition de domination stricte de la diagonale pour chaque ligne est : $a_{i,i}$ = 5*(i+1) >n-1

Pour de petites valeurs de n, l'élément diagonal $a_{i,i}$est bien plus grand que la somme des éléments hors-diagonaux. Par exemple pour n=3 : pour i=1 : on a 5>2, pour i=2 : on a 10>2 et pour i=3 : on a 15>2. Donc la condition est toujours vérifiée.

Cependant, plus n augmente, plus la différence entre l'élément diagonal et la somme des éléments hors-diagonaux est petite. Par exemple : si n=7, alors pour i=1 l'élément diagonal $a_{i,i}$ =5, et la somme des éléments hors-diagonaux dans chaque ligne est n-1=6. Donc la matrice n'est plus à diagonale dominante.

Convergence de la méthode de Jacobi :

La méthode de Jacobi converge si la matrice A est à diagonale dominante. Pour des petites tailles de n, la condition de domination stricte est vérifiée, donc la méthode de Jacobi converge rapidement.
Pour de plus grandes tailles de n, la domination devient moins évidente à mesure que la taille de la matrice augmente, donc la méthode de Jacobi converge plus lentement ou diverge.

Lien avec le rayon de convergence :

Le rayon de convergence est $\rho = \max(|\lambda_i|)$ où λ_i sont les valeurs propres de la matrice T. Plus ρ est petit, plus la méthode converge vite. Si $\rho \geq 1$, la méthode ne converge pas.
Pour que ρ soit proche de 0, il faut que les valeurs propres soient petites

C'est le cas lorsque A est une matrice à diagonale dominante.
En effet, si A est à diagonale dominante, alors les λ_i sont éloignées de 1 en valeur absolue, et ρ< 1. La convergence est rapide.
Si A n'est pas à diagonale dominante, ρ peut être proche de 1, ce qui ralentit la convergence ou supérieur à 1 et dans ce cas la méthode diverge.

Il sera donc important dans notre code de vérifier la domination de la diagonale et le rayon de convergence.

Nous avons programmé quatre programmes pour Jacobi dense en utilisant les différentes méthodes ci-dessous, puis comparé leurs performances :

Jacobi Method Convergence Investigation: (Minimum)
Expected outcome in file jacobi_dense.py

Première méthode (Jacobi_dense _avecinverse) en calculant avec la formule :
$\boldsymbol{x^{(k+1)} = D^{-1}(L+U)x^{(k)} + D^{-1} \cdot b}$

1) **Complexité**

Avec cette formule, nous utilisions le calcul direct de l'inverse de D (avec np.linalg.inv) qui a une complexité de O(n^3). C'est inutilement coûteux puisque D est une matrice diagonale, et son inverse pourrait être calculé directement en prenant les inverses des éléments diagonaux et la complexité serait alors de O(n).

2) **Vitesse de convergence**

La vitesse de convergence de Jacobi est relativement lente : dans le test avec n=100, 442 itérations sont nécessaires pour atteindre la tolérance. Le temps associé aux calculs est d'environ 1 seconde car le calcul de D−1 ajoute un coût important. Cela serait encore plus perceptible pour des dimensions plus grandes.

Nous allons donc créer une autre méthode afin d'éviter d'utiliser np.linalg.inv et remplacer par un calcul élément par élément.

Deuxième méthode (Jacobi_dense _boucleFor) en calculant avec la formule :

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

1) **Complexité**

La première boucle (for j in range(n)) parcourt tous les indices de x, ce qui donne une complexité de O(n).
La deuxième boucle calcule la somme de la partie supérieure et inférieure de chaque composant xj avec une complexité O(n).
Donc au total, pour chaque itération, la complexité O(n^2).

2) **Vitesse de convergence**

La vitesse de convergence de la méthode de Jacobi dépend du rayon spectral de la matrice d'itération. Si ce rayon est proche de 1, la méthode peut converger lentement. C'est ce que nous observons dans le test avec n=100, où 442 itérations sont nécessaires pour atteindre la tolérance.

Nous allons créer une nouvelle méthode afin de rendre les calculs plus rapides.

Troisième méthode (Jacobi_dense _somme) en calculant avec deux sommes (L et U) :

Cette méthode utilise la même logique que la méthode 2 mais nous avons vectorisé les sommes L et U pour améliorer l'efficacité des calculs. On ne calcule que les éléments nécessaires pour chaque itération ce qui minimise la quantité de calcul inutile. Cela rend l'algorithme plus rapide, en particulier pour des systèmes de grande taille.

1) **Complexité**

À chaque itération, nous effectuons une somme sur les éléments de L et U pour chaque xj. La complexité par itération est de O(n^2).

2) **Vitesse de convergence**

Avec n = 100, 450 itérations sont nécessaires pour atteindre la tolérance. Le nombre d'itération est donc plus important que pour la méthode 2 mais comme les calculs sont optimisés, le temps de calcul est réduit.

Quatrième méthode (Jacobi_dense _dot) en calculant L et U avec le produit scalaire :

1) **Complexité** :

Pour chaque élément j de la solution x on calcule L et U en faisant deux produits scalaires. Un produit scalaire pour L qui dépend des éléments avant l'élément j, et un pour U qui dépend des éléments après j. Chaque produit scalaire à une complexité de O(n) pour chaque élément j (donc pour n éléments), cela donne une complexité de O(n^2).

2) **Vitesse de convergence** :

La méthode avec produit scalaire est plus rapide que la méthode 3 grâce à l'utilisation des opérations vectorisées. Pour n=100 : Le temps d'exécution peut être de l'ordre de 0.2 à 1 seconde pour environ 400 itérations.

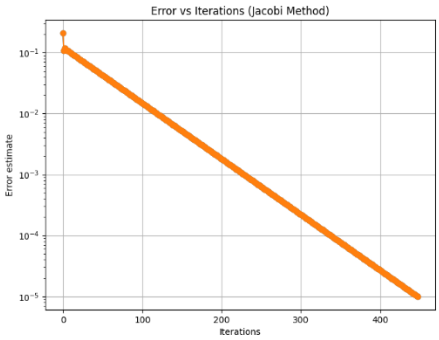
Nous avons résumé les résultats des quatre méthodes dans le tableau suivant :

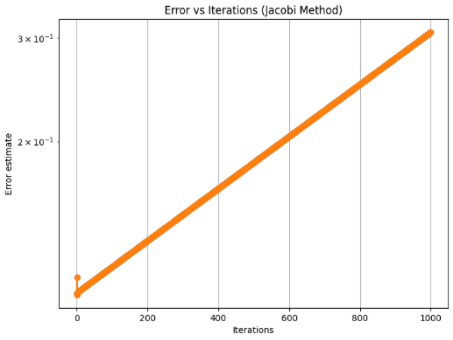
Temps d'itération pour n=	JD inverse	JD boucle	JD somme	JD dot
10	0,001059	0,000463	0.511357	0,229021
100	1,091545	0,981145	0.483927	0.208428
500	/	/	0,5004633	0,231787

Nous remarquons donc bien que les deux dernières méthodes ont un temps de calcul qui reste le même quelle que soit la taille de la matrice et qu'elles sont plus rapides pour des tailles de matrice élevées. En revanche les méthodes 1 et 2 sont plus rapides pour n=10, mais ont un temps de calcul qui augmente grandement lorsque n augmente.
De plus, les deux premières méthodes ne peuvent pas calculer les itérations pour des tailles de matrice trop grandes (n maximal = 110). Cela est du au rayon spectral de la matrice T.

Rayon spectral :

Nous avons implémenté une méthode permettant de calculer le rayon spectral. Le résultat est cohérent avec la théorie : lorsque $\rho < 1$, la méthode converge et si $\rho \geq 1$, elle diverge. Dans la méthode 2, la limite est atteinte pour n=110 : si n<111 $\rho < 1$, la méthode converge, et si n>111, $\rho \geq 1$, la méthode diverge.
Exemple (méthode 2) :





Pour n=100, on trouve $\rho = 0.97917067$ Pour n=111, on trouve $\rho = 1.0010234598$

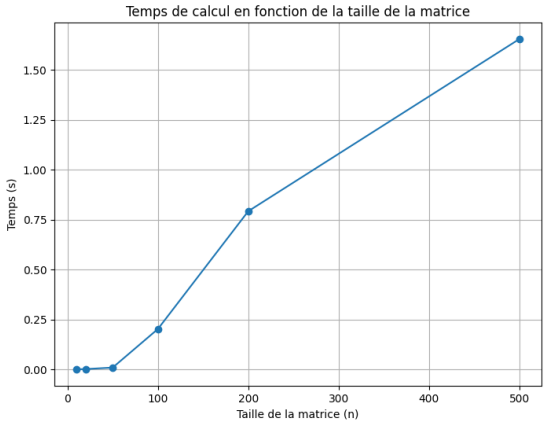
Cette fonction nous permet aussi de confirmer que pour cette matrice, plus n est petit, plus le rayon spectral est proche de 0. En effet, pour n=10, $\rho = 0.488322174$, pour n=25 $\rho = 0.68576450$, n=50 $\rho = 0.83323837$.

Diagonale dominante :

Nous avons également implémenté une méthode permettant de déterminer si la matrice est strictement diagonale dominante. On constate que pour la matrice $a_{i,i}$ = 5*(i+1) , pour i=1,….,n et $a_{i,j}$ = -1 pour i ≠ j, on a diagonale dominance jusqu'à n=6, ce qui confirme la théorie.
La matrice peut-être à diagonale non-dominante, tant que son rayon spectral est inférieur strict à 1, la méthode converge.

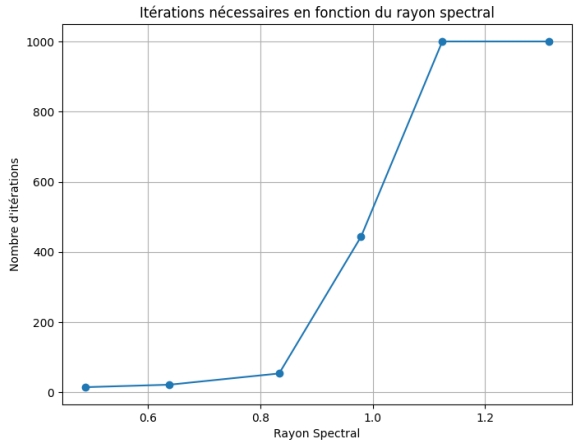
Plots (dans la méthode 4)

- Temps(t) en fonction de la taille de la matrice (n)



Nous observons bien que la taille de la matrice a un impact sur le temps de calcul : plus la matrice est grande, plus le temps de calcul est important. Dans le cas du produit scalaire (méthode 4), le temps varie très peu pour des petites dimensions et reste raisonnable même pour des dimensions importantes.

- Rayon spectral (ρ) en fonction du nombre d'itérations



Nous remarquons que plus le rayon spectral est proche de 1, plus le nombre d'itérations est important : le nombre d'itérations reste faible pour un rayon spectral inférieur à 0.85. Il augmente entre 0.85 et 1 mais reste fini donc la méthode converge. Cependant, si le rayon spectral est supérieur à 1, le nombre d'itérations est infini donc la méthode diverge

II) Méthode de Jacobi sur des matrices sparses

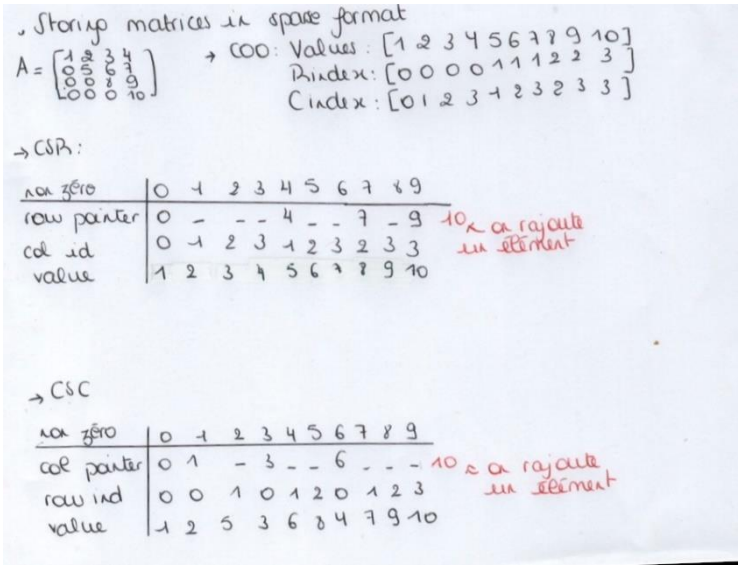
Pour augmenter encore l'efficacité de notre méthode, nous allons modifier le format de stockage des matrices en ne conservant que les éléments non-nuls. Pour cela, nous utilisons des matrices creuses (sparse matrix).

Il y a 3 formats différents pour cela :

- COO
- CSR
- CSC

Storing matrices in sparse format: Now you try

Pour bien comprendre comment les données sont stockées dans chaque format, nous nous sommes entrainées sur quelques exemples. Voici un scan de l'un d'entre eux :



Sparse and dense Jacobi: Now you try, file dense_sparse_jacobi.py

Nous avons implémenté cette méthode (Jacobi_sparse) et avons comparé les résultats avec Jacobi_dense

Nous avons fait une fonction qui crée une matrice tridiagonale creuse (generate_corrected_sparse_tridiagonal_matrix) à partir des valeurs de la diagonale principale et des sur et sous diagonales, ainsi qu'une version dense équivalente pour référence.
La fonction jacobi_sparse implémente la méthode de Jacobi sur des matrices creuses donc les calculs des diagonales et produits matriciels sont réduit.

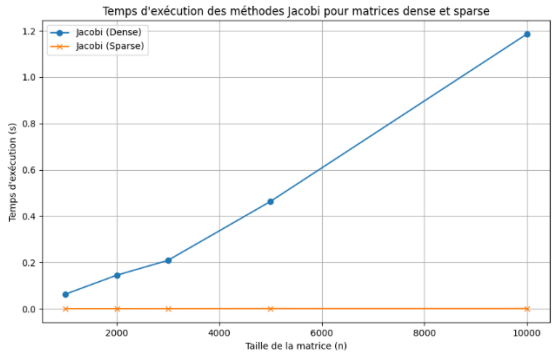
L'utilisation du format csr permet l'optimisation de la mémoire et du temps d'exécution.

Résultats (test):

n	Temps Jacobi_dense (sec)	Temps Jacobi_sparse (sec)
1000	0.0652	0.0002
2000	0.1117	0.0002
3000	0.2526	0.0004
5000	0.4387	0.0004
10000	1.1053	0.0006

Dans les deux méthodes, le nombre d'itérations est 18 pour n=1000, 2000, et 5000 et 19 pour n=5000 et 10000, ce qui indique qu'elles ont des performances similaires en termes de convergence. Cependant, la méthode Jacobi sparse est nettement plus rapide que la méthode Jacobi dense, et cet avantage devient de plus en plus prononcé avec l'augmentation de la taille n de la matrice.

Graphe :



III) Méthode de Gauss-Seidel

Now you try: file GS_jacobi_sparse.py

Préconditioner : C = D −L ou C = D −U

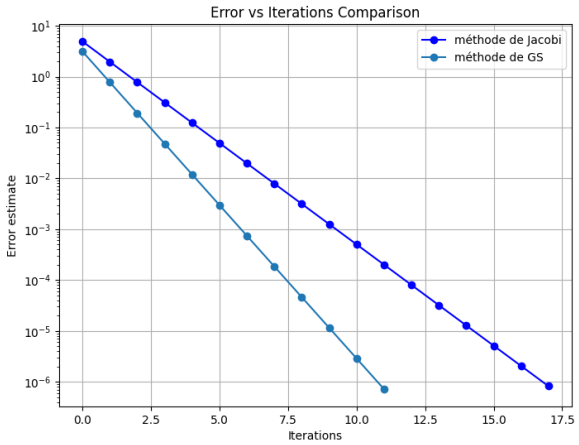
On applique ce préconditioner C au système Ax = b:

$$(D-L)^{-1}(D-U-L)x = (D+L)^{-1}b$$
$$(I-(D-L)^{-1}U)x = (D-L)^{-1}b$$
$$x = (D-L)^{-1}b + (D-L)^{-1}Ux$$

On implémente un méthode GS avec la forme itérative suivante :

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right) \quad \text{for } i = 1, 2, \ldots, n$$

Comparaison des résultats :



n	Temps Jacobi_sparse (sec)	Nb itérations Jacobi	Temps GS (sec)	Nb itérations GS
1000	0.0020	17	0.0272	12
2000	0.0203	18	0.1019	12
3000	0.0298	18	0.1066	12
5000	0.1739	18	0.2350	12

La méthode de Jacobi est plus rapide que la méthode de Gauss-Seidel pour toutes les tailles de matrice testées. En revanche, la méthode de Gauss-Seidel converge en moins d'itérations que Jacobi pour chaque taille de matrice donnée, son temps d'exécution par itération est donc plus élevé que celui de Jacobi.

Jacobi est plus rapide pour des matrices de petites à moyennes tailles, car le temps par itération est très faible. Cependant, elle nécessite plus d'itérations pour converger. Elle est donc plus adaptée aux matrices petites et moyennes.

Gauss-Seidel converge plus rapidement en termes de nombre d'itérations. Cependant, le coût par itération est plus élevé, ce qui signifie que, même avec moins d'itérations, le temps total peut augmenter pour des tailles de matrice plus grandes. Elle est donc plus adaptée si le nombre d'itérations maximal est petit.

IV) Méthode de SOR

Successive Over-Relaxation (SOR) Method

Pour finir, nous avons implémenter la méthode SOR, avec la formule itérative suivante :

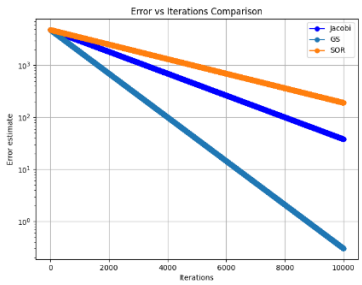
Preconditioner : $C = \frac{1}{\omega}(D - \omega L)$ Système préconditionné : $C^{-1}Ax = C^{-1}b$

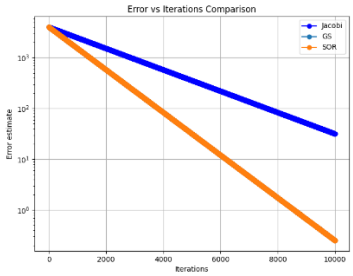
$$s_i^{(k)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$$
$$x_i^{(k+1)} = \omega s_i^{(k)} + (1-\omega)x_i^{(k)}$$

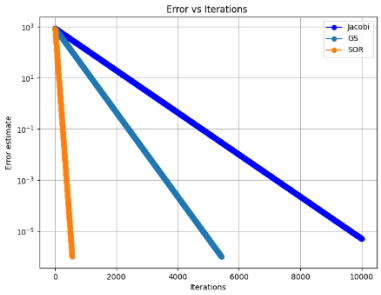
Cette méthode est similaire à la méthode de Gauss-Seidel, on introduit simplement un paramètre de relaxation ω compris entre 0 et 2 exclus. Ce facteur permet d'accélérer la convergence de l'algorithme.

Nous reprenons la méthode de GS, en modifiant 2 lignes de code : une ligne pour calculer $s_i^{(k)}$ et une ligne pour calculer $x_i^{(k+1)}$.

Résultats : On a pris pour tester la matrice avec des 2 sur la sur-diagonale et des -1 sur les sous/sur diagonales







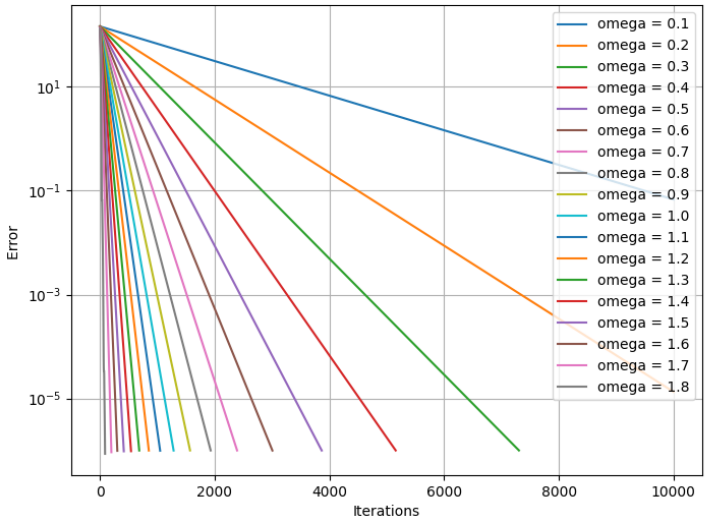
Pour ω =0.5 Pour ω =1 Pour ω =1.8

Nous observons que la vitesse de convergence dépend grandement du paramètre de relaxation ω :

- si ω=1, on se trouve dans le cas de la méthode GS
- si ω<1, la méthode SOR converge moins rapidement que les deux autres méthodes
- si ω >1, la méthode SOR converge plus rapidement que les deux autres méthodes

En testant avec différentes valeurs, on semble avoir une vitesse de convergence maximale pour ω = 1.8

Pour déterminer informatiquement le meilleur ω, nous codons une fonction (best_omega) qui teste des valeurs pour oméga entre 0.1 et 1.9 avec un pas de 0.1 et nous renvoyons la valeur de ω pour laquelle le nombre d'itérations est minimal.
Sur le graphique ci-dessous, nous pouvons voir les erreurs en fonction du nombre d'itérations pour les différentes valeurs de ω (pour une matrice de taille n=25).



Pour n=20, le meilleur ω est 1.8 pour la matrice avec des 2.5 sur la diagonale et des -1 sur la sous-diagonale et la sur-diagonale.

Le meilleur ω dépend aussi de la matrice A (matrice sparse)

Le principal avantage de cette méthode est qu'elle permet d'accélérer la convergence par rapport à Gauss-Seidel si ω est bien choisi. Néanmoins, il est nécessaire de déterminer le meilleur ω en faisant plusieurs essais et cela peut être long si on veut tester des valeurs précises de ω (avec un pas petit, par exemple 0.001).

Conclusion :

Durant ce projet, nous avons pu comprendre et expérimenter les différentes méthodes utilisées pour résoudre un système de type Ax=b en grande dimension. Il nous a permis de développer nos compétences en programmation Python et de mettre en évidence l'importance d'utiliser les bonnes méthodes en fonction du problème à traiter.