SYSTÈME LINÉAIRE EN GRANDE DIMENSION

Rapport

Années 2024-2025

Rédigé par :
CATTEAU Elsa
PROUZET Charlotte

MAM3

Introduction

Dans ce rapport de projet, nous allons considérer le système linéaire Ax=b

- A correspond à une matrice carrée de taille n * n
- b correspond à un vecteur de taille n * 1
- x est un vecteur de taille n * 1 et correspond à la solution de notre système linéaire

Afin de déterminer la valeur de x, nous allons implémenter différentes méthodes, afin de les comparer selon différents critères, à savoir :

- la valeur prise par x
- le nombre d'itérations
- les erreurs relatives à chaque itération
- la vitesse de convergence (via des graphes notamment)

En fonction des méthodes utilisées, nous pourrons également comparer de nouvelles variables de sorties. Dans le cas de la méthode de Jacobi dense, nous allons en effet étudier le rayon spectral.

I. Matrice large

La méthode de Jacobi repose sur la décomposition de la matrice A de notre système linéaire, en 3 autres matrices :

- D : la matrice diagonale
- L : la matrice triangulaire strictement inférieure
- U : la matrice triangulaire strictement supérieure

de sorte que A = D-L-U = D- (L+U)

A=

$$\begin{bmatrix} 3 & 0 & 4 \\ 7 & 4 & 2 \\ -1 & 1 & 2 \end{bmatrix}$$

D=

$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

U=

$$\begin{bmatrix} 0 & 0 & -4 \\ 0 & 0 & -2 \\ 0 & 0 & 0 \end{bmatrix}$$

L=

$$\begin{bmatrix} 0 & 0 & 0 \\ -7 & 0 & 0 \\ 1 & -1 & 0 \end{bmatrix}$$

Afin de résoudre notre système linéaire Ax=b, nous utilisons la décompositions de A sous la forme A=D-L-U et nous obtenons :
Ax=b
(D-L-U)x = b
Dx = b + (L+U)x
x = D^{-1}b + D^{-1}(L+U)x

La **forme Itérative** de Jacobi peut donc s'écrire :

$$x^{(k+1)} = D^{-1}(L+U)\, x^{(k)} + D^{-1}b$$

avec:

- $x^{(k)}$ est l'approximation actuelle du vecteur solution x à l'itération k
- $x^{(k+1)}$ est l'approximation mise à jour à l'itération k+1

Lors de l'implémentation de cette méthode sur Python, nous utiliserons dans un second temps la **forme par composant** qui est la suivante :

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

A) Définition

La méthode de Jacobi dense prend en argument la matrice A, le vecteur b, un vecteur x0 initial, une valeur dite de tolérance, et un nombre maximum d'itérations pour lequel, si atteint, la fonction s'arrête.

Cette fonction sera appelée :
jacobi_method(A, b, x0, tol=1e-5, max_iter=1000)

Dans celle-ci, nous utilisons la **forme par composante** de Jacobi, qui contient 3 boucles (for k, for i et for j)
voir annexe 1

Quand est-il de la convergence de la méthode ?

D'une part, une condition nécessaire et suffisante pour affirmer la convergence de la méthode repose sur l'étude du rayon spectral de la matrice T = D^{-1} (L+U).
On pose ρ(T) la valeur absolue de la valeur propre maximale de T.
Si ρ(T) < 1, alors il y a convergence de la méthode de Jacobi.

D'autre part, une condition suffisante pour affirmer la convergence serait de considérer la diagonale de la matrice A.
En effet, si A est strictement diagonalement dominante, alors la convergence de la méthode de Jacobi est garantie.

i.e : si $| a_{ii} | > \sum_{j\ !=\ i} | a_{ij} |$ pour tout i=1,2,...,n alors :
A est strictement diagonalement dominante

B) Jacobi_dense (3 boucles)

Afin de déterminer la convergence de la méthode de Jacobi, nous allons tester le code *annexe 2* qui nous apporte les informations suivantes:

- La matrice est-elle dominante ?
- Combien d'itérations ont-elles été réalisées ?
- Quelle est la solution de Jacobi trouvée ?
- Quelle est la solution exacte ?
- Quel est le rayon spectral ?

1) Premier cas:

A1=

$$\begin{bmatrix} 3 & 0 & 4 \\ 7 & 4 & 2 \\ -1 & 1 & 2 \end{bmatrix}$$

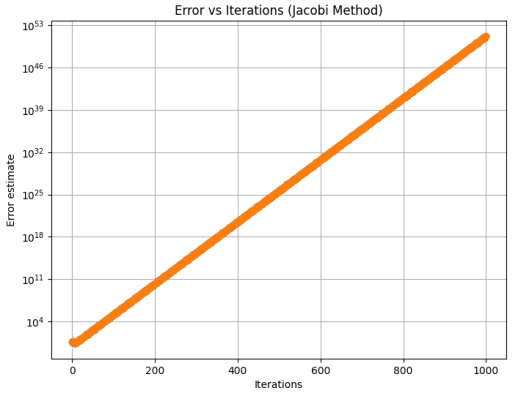
En posant A=np.array([3,0,4],[7,4,2],[-1,1,2]]) et x*=np.array([1,1,1]), on obtient b=Ax* = np.array([7,13,2])

En testant notre code python, nous apprenons que:

- La matrice A1 n'est pas dominante.
  >> en effet si l'on vérifie manuellement, on observe sur la 1ère ligne:
  3= |3 |< |0| + |4| = 4
- La solution trouvée est [-9.65e+50 7.53e+50 6.22e+49]
  >> donc il n'y a pas convergence.

- Le rayon spectral $\rho(T)$ est environ égal à 1.125
  >> $\rho(T) = 1.125 \geq 1$ donc on peut affirmer que certains points de départs n'arriveront pas à l'arrivée.

- On obtient le graphe représentant les erreurs en fonction du nombre d'itérations réalisées suivant:



>> Nous pouvons observer une courbe croissante. Ce résultat est cohérent puisque nous avons trouvé qu'il n'y a pas de convergence de Jacobi pour cette matrice-là.

2) Second cas:

A2=

$$\begin{bmatrix} -3 & 3 & -6 \\ -4 & 7 & -8 \\ 5 & 7 & -9 \end{bmatrix}$$

En posant A=np.array([-3,3,-9], [-4,7,-8], [5,7,-9]]) et x*=np.array([1,1,1]), on obtient: b=Ax* = np.array([-6,-5,3])

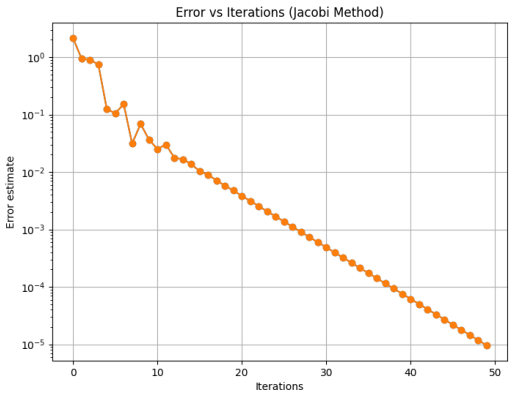
En testant notre code python, nous apprenons que:

- la matrice A2 n'est pas dominante
  >> en effet si l'on vérifie manuellement, on observe sur la 1ère ligne:
  3= | -3 |< |3| + |-6| = 9

- La solution trouvée est [0.999 1.000 0.999]
  >> donc il semble bien y avoir convergence.

- Le rayon spectral ρ(T) est environ égal à 0.813
  >> ρ(T) = 0.813 < 1 donc il y a bien convergence de Jacobi.

- On obtient le graphe représentant les erreurs en fonction du nombre d'itérations réalisées suivant:



>> Nous pouvons observer une courbe décroissante. Ce résultat est cohérent puisque nous avons montré qu'il y avait convergence de Jacobi pour la matrice A2.

C) Jacobi_dense2 (2 boucles)

Dans un second temps, de manière à alléger le code de la méthode Jacobi Dense, nous allons réutiliser **la forme par composante** de Jacobi (annexe 3). La seule différence par rapport à la première version de Jacobi dense est le nombre de boucles utilisées (on passe de 3 à 2 boucles).Pour ce faire, on utilise la fonction numpy.dot de manière à supprimer la troisième boucle générée par "for j".

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

$\sum a_{ij}\, x_j^{(k)}$ peut s'écrire np.dot(A[i , :] , x)

Ainsi : $\sum_{j\,!=\,i} a_{ij}\, x_j^{(k)} = \sum a_{ij}\, x_j^{(k)} - a_{ii}\, x_i^{(k)}$

$\sum_{j\,!=\,i} a_{ij}\, x_j^{(k)}$ = np.dot(A[i , :] , x) - (A[i , i] * x[i])

Donc $x_i^{(k+1)}$ = (1/ A[i , i]) * (b[i] - (np.dot(A[i , :] , x) - A[i , i] * x[i]))

$x_i^{(k+1)}$ = (1/ A[i , i]) * (b[i] - np.dot(A[i , :] , x) + A[i , i] * x[i])

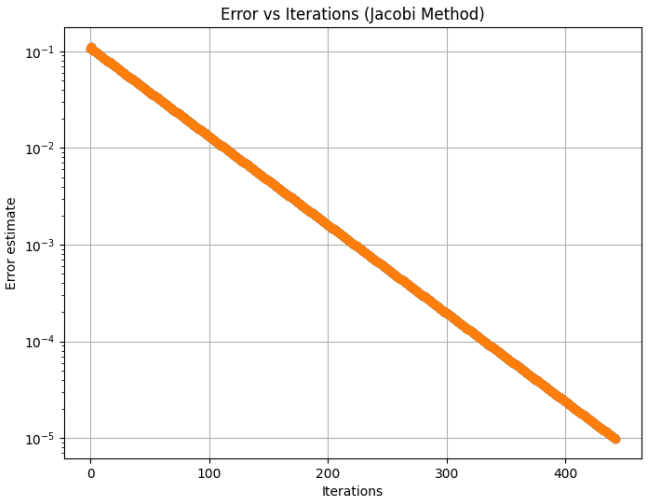
D) 3 boucles VS 2 boucles

Y a t-il une différence de temps et de nombre d'itération entre ces 2 méthodes ?
En comparant les codes (annexe 1) et (annexe 3), nous obtenons le tableau de résultats suivant :

	Nombre d'itérations	Temps d'exécution	Rayon Spectral
Jacobi dense avec 3 boucles	443	7.34s	0.979
Jacobi dense avec 2 boucles	446	0.54s	0.979

La forme par composante de Jacobi avec le .dot (2 boucles) est plus rapide que la forme initiale (3 boucles). Il sera donc plus judicieux de garder et réutiliser par la suite jacobi_dense2.

Dans les deux cas, le graphe représentant les erreurs de la valeur x en fonction du nombre d'itérations est le suivant :

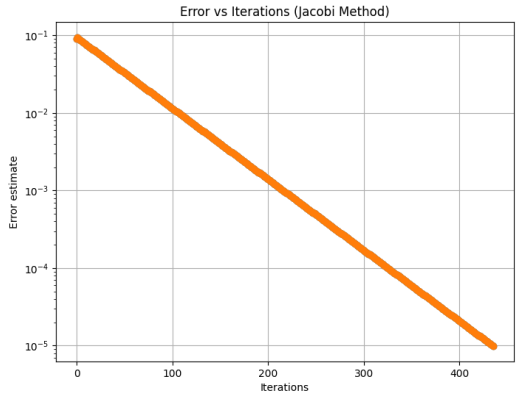


E) Rayon spectral - remarques

En testant notre second code Jacobi_dense2 (annexe3) nous pouvons donc facilement relever le rayon spectral de la matrice T et donc en déduire le comportement (convergence ou non) de la méthode.

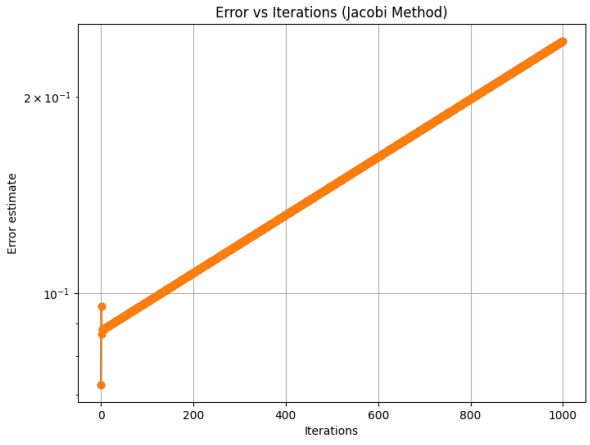
Au cours de nos essais, nous avons cependant relevé que le paramètre “n” correspondant à la taille de notre matrice A, joue un rôle prépondérant dans la valeur prise par le rayon spectral.

> En effet, en posant n=100, on obtient $\rho(T) = 0.979$
> En posant n=110, on obtient $\rho(T) = 0.999$



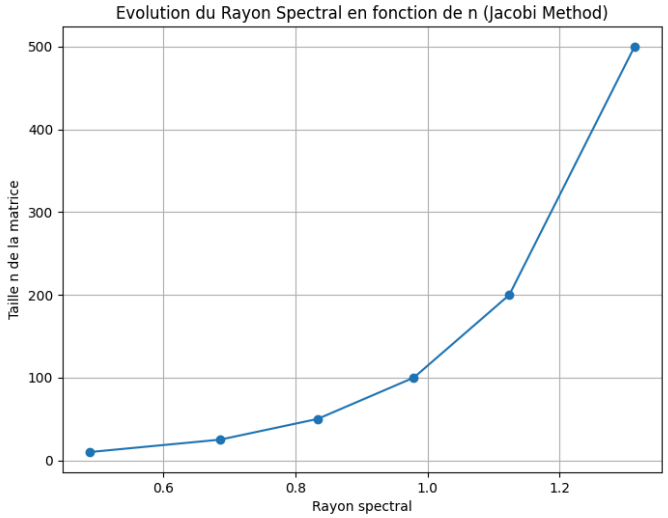
Il y a donc bien convergence de la méthode de Jacobi.

> En posant n=111, on obtient $\rho(T) = 1.001$
Le graphe représentant les erreurs de la valeur x en fonction du nombre d’itérations devient alors :



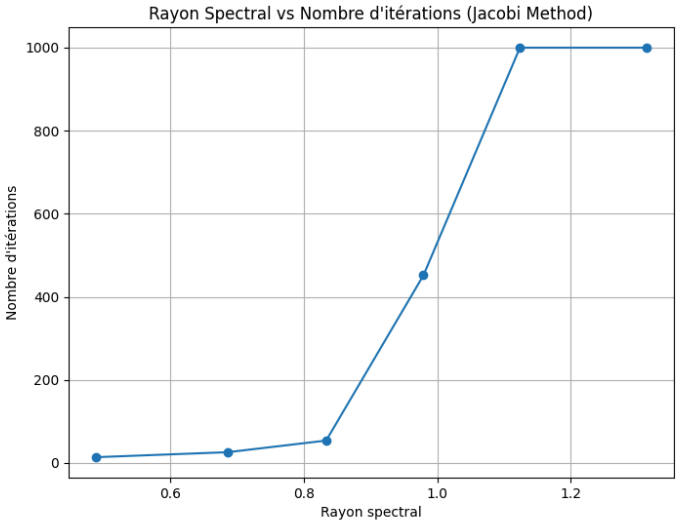
La courbe devient croissante, il n’y a donc plus convergence de la méthode de Jacobi.
De manière à gagner en visibilité, nous avons écrit un code (annexe 4 : jacobi_dense_comparaison_n_r) nous permettant de comparer l’évolution du rayon spectral par rapport:

1) à la taille de la matrice n:



>> Nous observons bien que le rayon spectral devient supérieur ou égal à 1 au bout de n=111 avec n la taille de notre matrice A.

2) au nombre d'itération (k+1):



>> Nous observons ici que le rayon spectral dépasse 1 au bout d'environ 540 itérations. Donc la méthode de Jacobi ne converge plus lorsque le nombre (k+1) d'itération est supérieur à 540. Cela correspondrait donc à une matrice de taille n*n avec n ≥ 111.

II- Matrices Creuses

Une matrice creuse est une matrice avec un nombre élevé d'éléments nuls, mais dont les seuls éléments pris en compte dans les calculs sont les éléments non nuls.

Une matrice tridiagonale est une matrice creuse dont les éléments non nuls se trouvent sur la diagonale, la diagonale et la sous-diagonale.

Il existe 3 formats de stockage pour les matrices creuses:

- CSR: Stocke dans un tableau les éléments non nuls de la matrice. Dans un autre tableau, il stocke le nombre d'éléments non nuls cumulés depuis l'entrée pour chaque ligne. Il stocke les indices des colonnes des éléments non nuls dans un troisième tableau.
- CSC: Similaire à CSR mais remplacer ligne par colonne.
- COO: Stock les indices des lignes et des colonnes des éléments non nuls et leur valeur

A) Jacobi Sparse

Pour Jacobi Sparse, nous n'avons pas pris la formule explicite de x_i. Nous sommes à la place passées par les matrices. En effet on a le système linéaire suivant : Ax = b

On a A= D-L-U, ce qui donne: (D-L-U)x = b

$$Dx = b+(L+U)x$$

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

La seule modification entre le programme de Jacobi dense et le programme de Jacobi Sparse est donc la formule de x.

Dans l'annexe 5, nous avons donc codé les 2 méthodes de Jacobi : Jacobi dense et Jacobi Sparse.
Nous avons ainsi cherché à comparer les 2 méthodes en fonction des critères suivants:

- nombre d'itérations
- temps d'exécution du programme

En posant n=100:

	Nombre d'itérations	Temps d'exécution
Jacobi Dense	17	0.0336s
Jacobi Sparse	19	0.0013s

En posant n=1000:

	Nombre d'itérations	Temps d'exécution
Jacobi Dense	18	0.4359s
Jacobi Sparse	20	0.0177s

On observe donc que la méthode de Jacobi Sparse est plus rapide que celle de Jacobi Dense.
Les nombres d'itérations sont quant à eux assez proches.

B) Préconditionnement

La méthode du préconditionnement est une méthode de résolution des systèmes linéaires de la forme Ax=b. On introduit une matrice C appelée « matrice de préconditionnement » qui est inversible tel que : $C^{-1}Ax = C^{-1}b$

C) Gauss Seidel

En posant: C = D-L

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

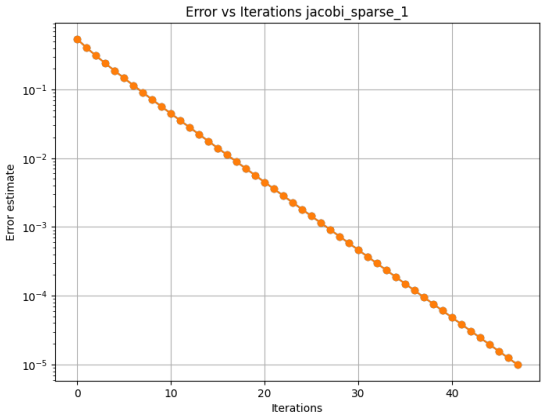
$(I - (D - L)^{-1}U)x = (D - L)^{-1}b$

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

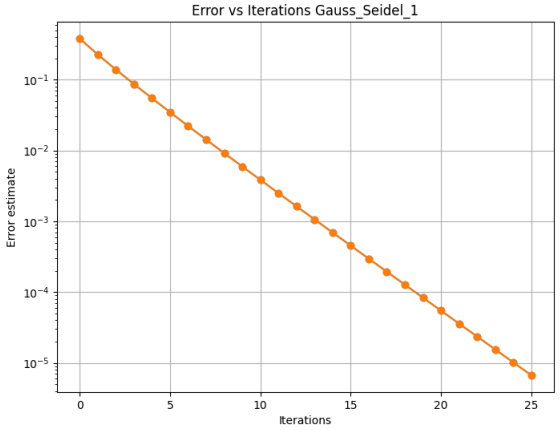
$x = C^{-1}(b + Ux)$

Afin d'appliquer la méthode Gauss Siedel, nous avons utilisé cette formule pour trouver x et nous avons affiché le graphe de l'erreur en fonction du nombre d'itérations.
(annexe 6)
On a d'abord testé la fonction generate_simple_sparse_tridiagonal_matrix qui génère une matrice tridiagonale en ayant la valeur de diagonale et de la sur/sous-diagonale avec. Nous avons obtenu les résultats (sous forme de graphes d'erreur) suivants :

1) par la méthode de Jacobi Sparse:



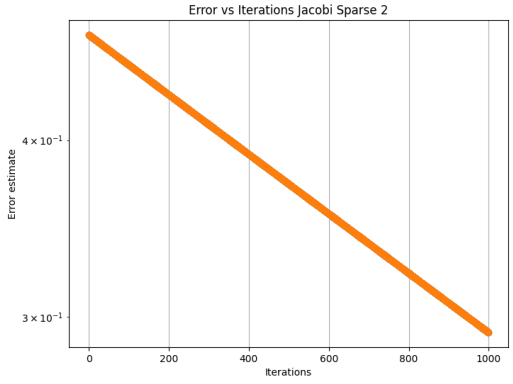
2) par la méthode de Gauss Seidel:



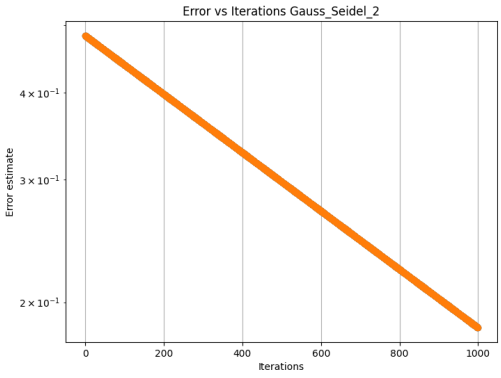
On a fait de même avec la fonction generate_sparse_tridiagonal_matrix qui utilise la méthode du Laplacien. Avec cette méthode on a le système linéaire Ax=B où $A = \frac{1}{h^2} * C$ où C est une matrice tridiagonale de taille n avec la valeur 2 sur la diagonale et -1 sur la sur/sous-diagonale, x=($f_1, f_2, .., f_n$) et b=($g_1, g_2, .., g_n$).

On obtient alors les graphes d'erreur suivants:

1) Pour la méthode Jacobi Sparse:



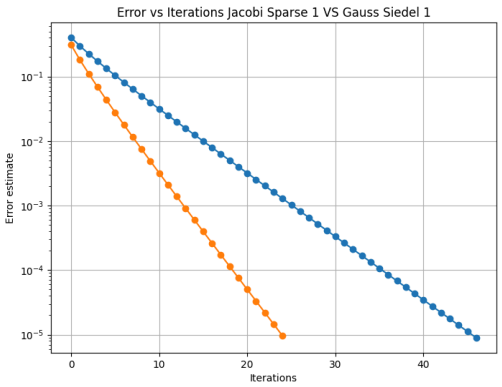
2) Pour la méthode avec Gauss Seidel

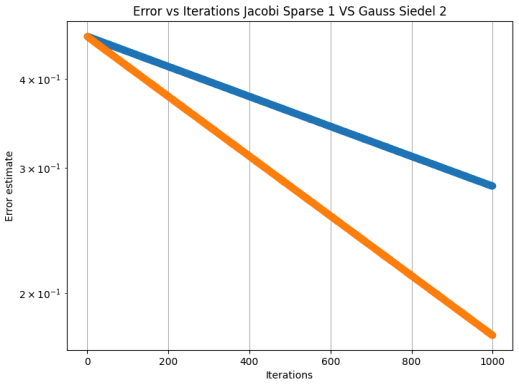


Que peut-on dire quant à la vitesse de convergence des deux méthodes ?

En superposant les graphiques de **Jacobi Sparse** et de **Gauss Seidel** pour chacune des 2 fonctions (generate_simple_sparse_tridiagonal_matrix et generate_sparse_tridiagonal_matrix), nous obtenons les courbes suivantes:

(le code se trouve en annexe (annexe 7))





On observe que la méthode Gauss Seidel a une vitesse de convergence supérieure à celle de la méthode Jacobi Sparse.

Que peut-on dire quant au temps d'exécution de chacune de ces deux méthodes ?

Pour chaque méthode, nous avons répertorié les résultats obtenus dans le tableau suivant :

	Jacobi Sparse	Gauss Seidel
generate_simple_sparse_tridiagonal_matrix	0.0175s	0.1136s
generate_sparse_tridiagonal_matrix	0.0601s	0.2383s

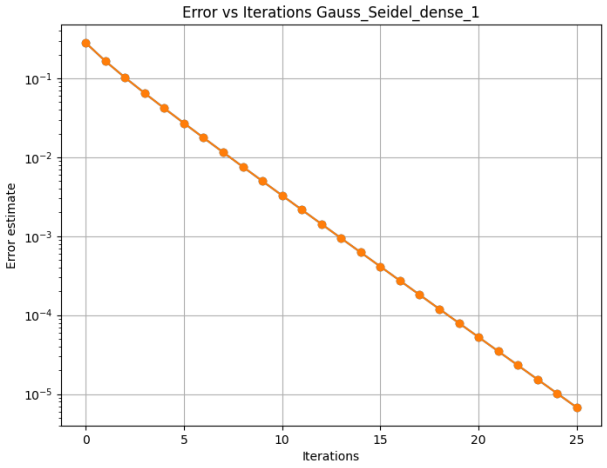
Dans les deux cas, le temps d'exécution de la méthode de Gauss Seidel est plus long que celui de la méthode de Jacobi.

Pour résumer, la méthode de Gauss Siedel a une vitesse de convergence plus rapide que celle de Jacobi Sparse. En revanche, son temps d'exécution est supérieur. Ce résultat est plutôt surprenant.

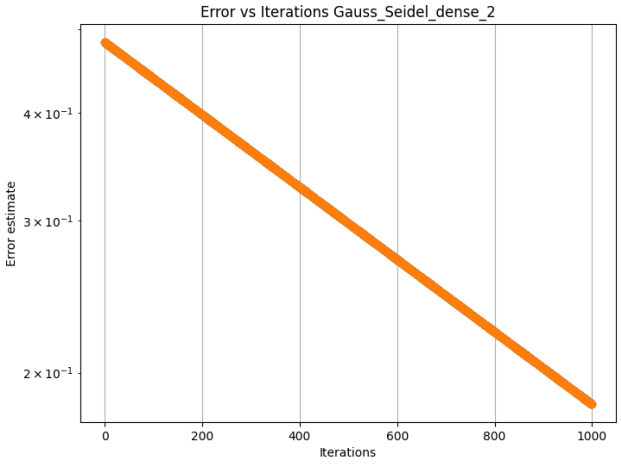
Remarque

On peut également remarquer qu'en faisant une fonction Gauss_Seidel_dense qui prend en paramètre une matrice dense, on obtient les mêmes graphes qu'avec la matrice sparse. **(ANNEXE ??)**

Avec la fonction generate_simple_sparse_tridiagonal_matrix on obtient:



Avec la fonction generate_sparse_tridiagonal_matrix on obtient:



Ensuite, on a également ajouté le calcul du rayon spectral dans la fonction de Jacobi Sparse et de Gauss Seidel. Pour cela, on a adapté le code fait dans le fichier jacobi

dense pour que cela fonctionne avec une matrice sparse. On a donc réutilisé la formule de T=$D^{-1}(L+U)$ mais cette fois-ci T est une matrice sparse. Pour pouvoir utiliser le fait que le rayon spectral est le maximum des valeurs absolues des valeurs propres on a transformé T en matrice dense en mettant T.todense.

Comparaison des matrices A3 et A4 avec Jacobi dense et Gauss Seidel:

$$A_3 = \begin{bmatrix} 4 & 1 & 1 \\ 2 & -9 & 0 \\ 0 & -8 & -6 \end{bmatrix} \quad A_4 = \begin{bmatrix} 7 & 6 & 9 \\ 4 & 5 & -4 \\ -7 & -3 & 8 \end{bmatrix}$$

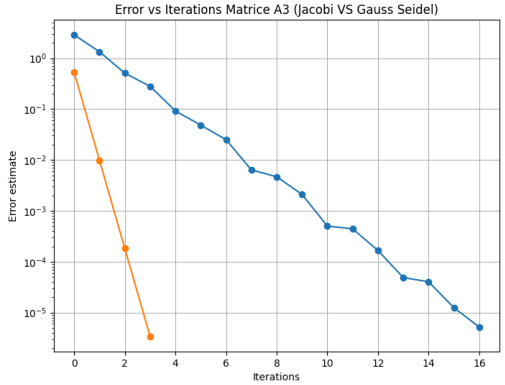
Pour cela, on a réutilisé la fonction jacobi_method et gauss_seidel_dense_with_error. **(ANNEXE 10)**

	Jacobi	Gauss Seidel
Temps A3	0.0003616809844970703 s	0.00012683868408203125 s
Temps A4	0.0004639625549316406 s	0.0003654956817626953 s

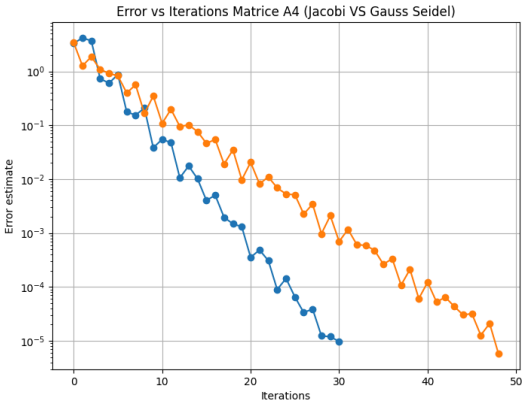
	Jacobi	Gauss Seidel
Itérations A3	17	4
Itérations A4	31	49

On obtient donc les graphes d'erreurs et d'itérations pour les méthodes de **Jacobi Sparse** et **Gauss Seidel** suivants:

Pour la matrice A3:



Pour la matrice A4:



III- Méthode SOR

La méthode SOR peut être considérée comme une méthode itérative préconditionnée avec C la matrice de préconditionnement et ω le paramètre de relaxation tel que $0<\omega<2$.

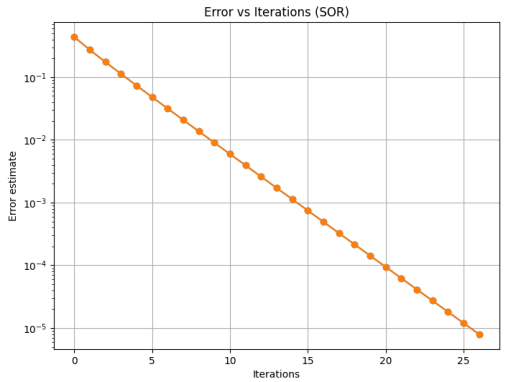
On pose C = $\frac{1}{\omega}(D - \omega L)$.

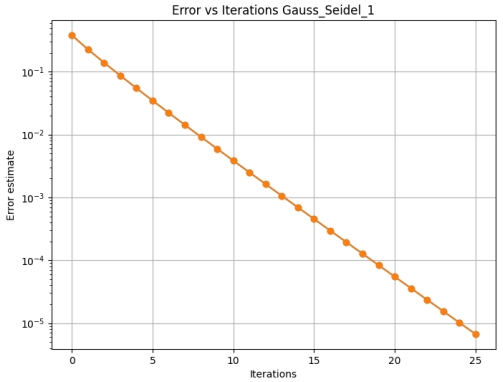
Pour trouver la formule de x, on résout :

$$C^{-1}Ax = C^{-1}b$$
$$C^{-1}(D - L - U)x = C^{-1}b$$
$$(\frac{D}{\omega} - L)^{-1}(\frac{D}{\omega} - L + D - \frac{D}{\omega})x = C^{-1}b$$
$$(I + (\frac{D}{\omega} - L)^{-1}(D - U - \frac{D}{\omega}))x = C^{-1}b$$
$$(I + C^{-1}(D(I - \frac{1}{\omega}) - U)x = C^{-1}b$$
$$x = C^{-1}b - C^{-1}(\frac{\omega-1}{\omega}D - U)x$$
$$x = C^{-1}(b - (\frac{\omega-1}{\omega}D - U)x)$$

Pour générer notre matrice, on a utilisé la fonction generate_sparse_tridiagonal_matrix faite dans le fichier de la méthode de Gauss Seidel. Ensuite on a créé une fonction successive_over_relaxation qui prend en argument une matrice sparse A, b, x0, x_exact, tol=1e^-5 et w (qui correspond à ω. On a repris la même forme que la fonction qui implémente Gauss Sideil mais on a définit C=(1/w)*(D+w*L) et on a changé la formule de x_new en mettant x_new = C1.dot(b-(((w-1)/w)*D+U).dot(x)), qui correspond à la formule que nous avons trouvé mathématiquement.

(annexe 8 SOR)



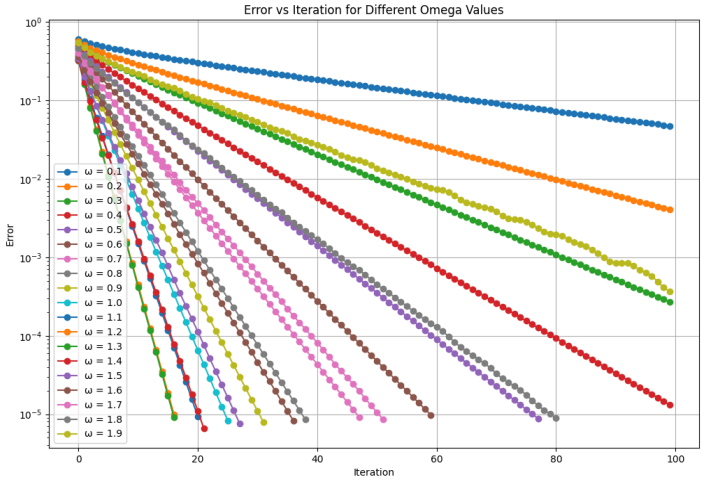


Avec $\omega = 1$, on retrouve le même graphe que pour la méthode Gauss Seidel.

Ensuite, nous avons voulu déterminer quel est le meilleur oméga, c'est-à-dire celui qui converge le plus rapidement. Pour cela, nous avons créé une fonction best_omega(As, b,x0,x_exact, tol=1e-5, max_iter=100) qui utilise la fonction Sucessive_Over_Relaxation. Grâce à cette fonction on obtient le nombre d'itération et on cherche pour quel omega le nombre d'itérations est minimal. La fonction renvoie donc le meilleur omega. Le programme nous donne que le meilleur omega est 1.2.

On a ensuite décidé de créer une fonction qu'on a appelé plot_error_omegas(As, b, x0, x_exact, tol=1e-5, max_iter=100), qui trace le graphe du nombre d'itérations en fonction de l'erreur et où l'on peut voir la convergence pour chaque omega. Pour cela, on crée un tableau qui commence à 0.1 jusqu'à 1.9 avec un pas de 0.1, pour cela on utilise l'instruction np.arange(0.1, 2.0, 0.1). Cette fonction utilise également la fonction Sucessive_Over_Relaxation, ce qui nous donne accès au tableau d'erreurs pour chaque omega.
(annexe 9)

On obtient le graphe ci-dessous



A l'aide de ce graphe, on observe donc que la vitesse de convergence est la plus optimale pour omega = 1.2
Ce résultat est cohérent avec la valeur du "best_omega" générée par la fonction portant le même nom.

Conclusion ?

ANNEXES

1) Jacobi_dense

```
import numpy as np
import matplotlib.pyplot as plt
import time

def generate_linear_system(n):
  """
  Generates a linear system with a diagonally dominant matrix A
and vector b.

  Args:
    n: Dimension of the system (size of the matrix A).

  Returns:
    A: Coefficient matrix (numpy array).
    b: Right-hand side vector (numpy array).
  """

  A = (-1)*np.ones((n, n))
  for i in range(n):
    A[i,i]  = 5*(i+1)

  b = np.random.rand(n)

  return A, b

# Example usage:
n = 100  # Dimension of the system
A, b = generate_linear_system(n)
#print(A)
#print(b)


def jacobi_method(A, b, x0, tol=1e-5, max_iter=1000):
  """
  Implements the Jacobi method for solving the linear system Ax =
b.

  Args:
    A: Coefficient matrix (numpy array).
    b: Right-hand side vector (numpy array).
```

```
    x0: Initial guess for the solution vector (numpy array).
    tol: Tolerance for convergence.
    max_iter: Maximum number of iterations.

  Returns:
    x: Approximate solution vector.
    iterations: Number of iterations performed.
    errors: List of errors between exact and approximate solution
at each iteration.
  """

  start_time = time.time()

  n = A.shape[0]
  x = x0.copy()
  errors = []
  for k in range(max_iter):
    x_new=np.zeros_like(x)
    for i in range(n):
      sum=0
      for j in range(n):
        if j!=i:
          sum=sum+A[i,j]*x[j]
      x_new[i]=(1/A[i,i]) * (b[i] - sum)
    error = np.linalg.norm(x_new - x)
    errors.append(error)
    x = x_new
    if error < tol:
      break

  #calcul du rayon spectral
  D=np.diag(np.diag(A))
  D2=np.linalg.inv(D) #correspond à l'inverse de D
  T=np.matmul(D2, D-A) #matrice dont on cherche les valeurs
propres pour trouver le rayon spectral
  r=np.max(np.absolute(np.linalg.eigvals(T))) #si r<1 alors
convergence de la méthode de Jacobi

  #A est-elle une matrice dominante ?
  if np.all(((np.absolute(np.diag(A)))) >
np.sum(np.absolute(A-D), axis=1)): #axis=1 permet de vérifier la
conditon colonne par colonne
    print("La matrice A est dominante")
```

```
  else:
    print("la matrice A n'est pas dominante")

  end_time=time.time()
  time_taken = end_time - start_time

  return x, k + 1, errors, time_taken, r

def plot_error(errors, iterations):
    plt.figure(figsize=(8, 6))
    plt.plot(range(iterations), errors, marker='o',
linestyle='-')
    plt.semilogy(range(iterations), errors, marker='o',
linestyle='-')  # Use semilogy for log-scale on y-axis
    plt.xlabel("Iterations")
    plt.ylabel("Error estimate")
    plt.title("Error vs Iterations (Jacobi Method)")
    plt.grid(True)
    plt.show()



# Example usage:
n = 100
A, b = generate_linear_system(n)  # Generate a linear system
x0 = np.zeros(np.size(b))

# Solve using Jacobi method (avec le temps associé)
x_jacobi, iterations, errors, time, rayon_spectral =
jacobi_method(A, b, x0)

# Calculate exact solution
x_exact = np.linalg.solve(A, b)

# Print results
print(f"Iterations: {iterations}")
print(f"Solution Jacobi: {x_jacobi}")
print(f"Exact solution: {x_exact}")
print(f"Rayon Spectral : {rayon_spectral}")
print(f"Time taken : {time}") #environ 7sec pour la méthode avec
3 boucles

# Plot the error
plot_error(errors, iterations)
```

2) jacobi_dense_verif

```
#pour vérifier les résultats obtenus à la main pour A0

import numpy as np
import matplotlib.pyplot as plt

def jacobi_method(A, b, x0, tol=1e-5, max_iter=1000):
  """
  Implements the Jacobi method for solving the linear system Ax = b.

  Args:
    A: Coefficient matrix (numpy array).
    b: Right-hand side vector (numpy array).
    x0: Initial guess for the solution vector (numpy array).
    tol: Tolerance for convergence.
    max_iter: Maximum number of iterations.

  Returns:
    x: Approximate solution vector.
    iterations: Number of iterations performed.
    errors: List of errors between exact and approximate solution at
each iteration.
  """

  n = A.shape[0]
  x = x0.copy()
  errors = []
  for k in range(max_iter):
    x_new=np.zeros_like(x)
    for i in range(n):
      sum=0
      for j in range(n):
        if j!=i:
          sum=sum+A[i,j]*x[j]
      x_new[i]=(1/A[i,i]) * (b[i] - sum)
    error = np.linalg.norm(x_new - x)
    errors.append(error)
    x = x_new
    if error < tol:
      break

  #calcul du rayon spectral
```

```
  D=np.diag(np.diag(A))
  D2=np.linalg.inv(D) #correspond à l'inverse de D
  T=np.matmul(D2, D-A) #matrice dont on cherche les valeurs propres
pour trouver le rayon spectral
  r=np.max(np.absolute(np.linalg.eigvals(T))) #si r<1 alors convergence
de la méthode de Jacobi

  #A est-elle une matrice dominante ?
  if np.all(((np.absolute(np.diag(A)))) > np.sum(np.absolute(A-D),
axis=1)): #axis=1 permet de vérifier la conditon colonne par colonne
    print("La matrice A est dominante")
  else:
    print("la matrice A n'est pas dominante")

  return x, k + 1, errors, r

def plot_error(errors, iterations):
    plt.figure(figsize=(8, 6))
    plt.plot(range(iterations), errors, marker='o', linestyle='-')
    plt.semilogy(range(iterations), errors, marker='o', linestyle='-')
# Use semilogy for log-scale on y-axis
    plt.xlabel("Iterations")
    plt.ylabel("Error estimate")
    plt.title("Error vs Iterations (Jacobi Method)")
    plt.grid(True)
    plt.show()


# Example usage:
A=np.array([[2,-1],[-1,2]])
b=np.array([1, 1])
x0=np.zeros(len(b)) #ou np.size(b)

# Solve using Jacobi method
x_jacobi, iterations, errors, rayon_spectral = jacobi_method(A, b, x0)

# Calculate exact solution
x_exact = np.linalg.solve(A, b)

# Print results
print(f"Iterations: {iterations}")
print(f"Solution Jacobi: {x_jacobi}")
print(f"Exact solution: {x_exact}")
```

```
print(f"Rayon spectral: {rayon_spectral}")

# Plot the error
plot_error(errors, iterations)
```

3) jacobi_dense2

```
import numpy as np
import matplotlib.pyplot as plt
import time

def generate_linear_system(n):
  """
  Generates a linear system with a diagonally dominant matrix A and
vector b.

  Args:
    n: Dimension of the system (size of the matrix A).

  Returns:
    A: Coefficient matrix (numpy array).
    b: Right-hand side vector (numpy array).
  """

  A = (-1)*np.ones((n, n))
  for i in range(n):
    A[i,i]  = 5*(i+1)

  b = np.random.rand(n)

  return A, b

# Example usage:
n = 100  # Dimension of the system
A, b = generate_linear_system(n)
#print(A)
#print(b)



def jacobi_method(A, b, x0, tol=1e-5, max_iter=1000):
  """
  Implements the Jacobi method for solving the linear system Ax = b.
```

```
  Args:
    A: Coefficient matrix (numpy array).
    b: Right-hand side vector (numpy array).
    x0: Initial guess for the solution vector (numpy array).
    tol: Tolerance for convergence.
    max_iter: Maximum number of iterations.

  Returns:
    x: Approximate solution vector.
    iterations: Number of iterations performed.
    errors: List of errors between exact and approximate solution at
each iteration.
  """

  start_time=time.time()

  n = A.shape[0]
  x = x0.copy()
  errors = []
  for k in range(max_iter):
    x_new=np.zeros_like(x)
    for i in range(n):
        x_new[i]=(1/A[i,i]) * (b[i] - np.dot(A[i,:],x) + A[i,i]*x[i])
#on utilise le produit scalaire
    error = np.linalg.norm(x_new - x)
    errors.append(error)
    x = x_new
    if error < tol:
      break

  #calcul du rayon spectral
  D=np.diag(np.diag(A))
  D2=np.linalg.inv(D) #correspond à l'inverse de D
  T=np.matmul(D2, D-A) #matrice dont on cherche les valeurs propres
pour trouver le rayon spectral
  r=np.max(np.absolute(np.linalg.eigvals(T))) #si r<1 alors convergence
de la méthode de Jacobi

  #A est-elle une matrice dominante ?
  if np.all(((np.absolute(np.diag(A)))) > np.sum(np.absolute(A-D),
axis=1)): #axis=1 permet de vérifier la conditon colonne par colonne
    print("La matrice A est dominante")
```

```
  else:
    print("la matrice A n'est pas dominante")

  end_time = time.time()
  time_taken = end_time - start_time

  return x, k + 1, errors, time_taken, r

def plot_error(errors, iterations):
    plt.figure(figsize=(8, 6))
    plt.plot(range(iterations), errors, marker='o', linestyle='-')
    plt.semilogy(range(iterations), errors, marker='o', linestyle='-')
# Use semilogy for log-scale on y-axis
    plt.xlabel("Iterations")
    plt.ylabel("Error estimate")
    plt.title("Error vs Iterations (Jacobi Method)")
    plt.grid(True)
    plt.show()


# Example usage:
n = 100
A, b = generate_linear_system(n)  # Generate a linear system
x0 = np.zeros(np.size(b))

# Solve using Jacobi method
x_jacobi, iterations, errors, time, rayon_spectral = jacobi_method(A,
b, x0)

# Calculate exact solution
x_exact = np.linalg.solve(A, b)

# Print results
print(f"Iterations: {iterations}")
print(f"Solution Jacobi: {x_jacobi}")
print(f"Exact solution: {x_exact}")
print(f"Rayon Spectral : {rayon_spectral}")
print(f"Time (avec produit scalaire): {time}") #environ 0.49sec pour la
méthode avec le produit scalaire (elle est donc beaucoup plus rapide
que celle avec la boucle j!)

# Plot the error
plot_error(errors, iterations)
```

4) jacobi_dense_comparaison_n_r

```
import numpy as np
import matplotlib.pyplot as plt
import time

def generate_linear_system(n):
  """
  Generates a linear system with a diagonally dominant matrix A
and vector b.

  Args:
    n: Dimension of the system (size of the matrix A).

  Returns:
    A: Coefficient matrix (numpy array).
    b: Right-hand side vector (numpy array).
  """

  A = (-1)*np.ones((n, n))
  for i in range(n):
    A[i,i]  = 5*(i+1)

  b = np.random.rand(n)

  return A, b

# Example usage:
n = 100  # Dimension of the system
A, b = generate_linear_system(n)
#print(A)
#print(b)


def jacobi_method(A, b, x0, tol=1e-5, max_iter=1000):
  """
  Implements the Jacobi method for solving the linear system Ax =
b.

  Args:
    A: Coefficient matrix (numpy array).
    b: Right-hand side vector (numpy array).
    x0: Initial guess for the solution vector (numpy array).
```

```
    tol: Tolerance for convergence.
    max_iter: Maximum number of iterations.

  Returns:
    x: Approximate solution vector.
    iterations: Number of iterations performed.
    errors: List of errors between exact and approximate solution
at each iteration.
  """

  start_time=time.time()

  n = A.shape[0]
  x = x0.copy()
  errors = []
  for k in range(max_iter):
    x_new=np.zeros_like(x)
    for i in range(n):
        x_new[i]=(1/A[i,i]) * (b[i] - np.dot(A[i,:],x) +
A[i,i]*x[i]) #on utilise le produit scalaire
    error = np.linalg.norm(x_new - x)
    errors.append(error)
    x = x_new
    if error < tol:
      break

  #calcul du rayon spectral
  D=np.diag(np.diag(A))
  D2=np.linalg.inv(D) #correspond à l'inverse de D
  T=np.matmul(D2, D-A) #matrice dont on cherche les valeurs
propres pour trouver le rayon spectral
  r=np.max(np.absolute(np.linalg.eigvals(T))) #si r<1 alors
convergence de la méthode de Jacobi

  #A est-elle une matrice dominante ?
  if np.all(((np.absolute(np.diag(A)))) >
np.sum(np.absolute(A-D), axis=1)): #axis=1 permet de vérifier la
conditon colonne par colonne
    print("La matrice A est dominante")
  else:
    print("la matrice A n'est pas dominante")

  end_time = time.time()
```

```
  time_taken = end_time - start_time

  return x, k + 1, errors, time_taken, r

def test():
    n_tab=[]
    r_tab=[]
    iter_tab=[]
    for n in [10, 25 , 50, 100, 200, 500]:
        A, b = generate_linear_system(n)  # Generate a linear
system
        x0 = np.zeros(np.size(b))
        x_jacobi, iterations, errors, time, rayon_spectral =
jacobi_method(A, b, x0)
        n_tab.append(n)
        r_tab.append(rayon_spectral)
        iter_tab.append(iterations)
    print(f"taille de la matrice {n_tab}, rayon spectral {r_tab},
{iter_tab} itérations")
    return n_tab, r_tab, iter_tab

def plot_1(r_tab, n_tab):
    plt.figure(figsize=(8, 6))
    plt.plot(r_tab, n_tab, marker='o', linestyle='-')
    plt.xlabel("Rayon spectral")
    plt.ylabel("Taille n de la matrice")
    plt.title("Evolution du Rayon Spectral en fonction de n
(Jacobi Method)")
    plt.grid(True)
    plt.show()

def plot_2(r_tab, iter_tab):
    plt.figure(figsize=(8, 6))
    plt.plot(r_tab, iter_tab, marker='o', linestyle='-')
    plt.xlabel("Rayon spectral")
    plt.ylabel("Nombre d'itérations")
    plt.title("Rayon Spectral vs Nombre d'itérations (Jacobi
Method)")
    plt.grid(True)
    plt.show()


n_tab, r_tab, iter_tab =test()
```

```
plot_1(r_tab, n_tab) #graphe de l'évolution du rayon spectral en
fonction de la taille n de la matrice
plot_2(r_tab, iter_tab) #graphe de l'évolution du rayon spectral
en fonction du nombre d'itérations réalisées
```

5) dense_sparse_jacobi

```
import numpy as np
import scipy.sparse as sparse
from scipy.sparse import csr_matrix
import time


def generate_corrected_sparse_tridiagonal_matrix(n,
diagonal_value=5, off_diagonal_value=1):
    """
    Generates a sparse tridiagonal matrix, ensuring no overlaps.

    Args:
        n: Dimension of the system (size of the matrix A).
        diagonal_value: Value for the diagonal elements.
        off_diagonal_value: Value for the off-diagonal elements.

    Returns:
        A: Sparse coefficient matrix (scipy.sparse.csr_matrix).
        b: Right-hand side vector (numpy array).
    """

    # Main diagonal
    main_diag = np.full(n, diagonal_value) #vecteur de taille
(1,n) ne contenant que des 5 (valeur de la diagonale)
    off_diag = np.full(n-1,off_diagonal_value) #vecteur de taille
(1,n-1) ne contenant que des 1 (valeur de la sur-diagonale et de
la sous-diagonale)

    # Construct sparse matrix
    data = np.concatenate([main_diag, off_diag, off_diag])
    print(data)
    rows = np.concatenate([np.arange(n), np.arange(n-1),
np.arange(1,n)])
    print(rows)
```

```
    cols = np.concatenate([np.arange(n), np.arange(1,n),
np.arange(n-1)])
    print(cols)
    As = csr_matrix((data, (rows, cols)), shape=(n, n))
    print(As)

    # Construct dense matrix for reference
    A_dense = np.zeros((n, n))
    A_dense[n-1,n-1]=diagonal_value
    for i in range(n-1):
        A_dense[i, i] = diagonal_value
        A_dense[i, i+1]= off_diagonal_value
        A_dense[i+1,i]=off_diagonal_value

    b = np.random.rand(n)
    return As, A_dense, b


def jacobi_dense(A, b, x0, tol=1e-6, max_iter=1000):
    """
    Jacobi method for dense matrices.

    Args:
        A: Dense coefficient matrix (numpy array).
        b: Right-hand side vector (numpy array).
        x0: Initial guess for the solution vector (numpy array).
        tol: Tolerance for convergence.
        max_iter: Maximum number of iterations.

    Returns:
        x: Approximate solution vector.
        iterations: Number of iterations performed.
        time_taken: Time taken for the iterations.

    """

    start_time=time.time()

    n = A.shape[0]
    x = x0.copy()
    errors = []
    for k in range(max_iter):
        x_new=np.zeros_like(x)
```

```
        for i in range(n):
            x_new[i]=(1/A[i,i]) * (b[i] - np.dot(A[i,:],x) +
A[i,i]*x[i]) #on utilise le produit scalaire
        error = np.linalg.norm(x_new - x)
        errors.append(error)
        x = x_new
        if error < tol:
            break

    end_time = time.time()
    time_taken = end_time - start_time

    return x, k + 1, time_taken


def jacobi_sparse(A, b, x0, tol=1e-7, max_iter=10000):
    """
    Jacobi method for sparse matrices.

    Args:
        A: Sparse coefficient matrix (scipy.sparse.csr_matrix).
        b: Right-hand side vector (numpy array).
        x0: Initial guess for the solution vector (numpy array).
        tol: Tolerance for convergence.
        max_iter: Maximum number of iterations.

    Returns:
        x: Approximate solution vector.
        iterations: Number of iterations performed.
        time_taken: Time taken for the iterations.
    """
    start_time=time.time()

    x = x0.copy()
    D1=1/A.diagonal()
    LU=A-sparse.diags(A.diagonal()) #correspond à -(L+U)

    for k in range(max_iter):
        x_new=D1*(b-LU.dot(x))

        error = np.linalg.norm(x_new - x)

        x = x_new
```

```
        if error < tol:
            break

    end_time = time.time()
    time_taken = end_time - start_time

    return x, k+1, time_taken

# Example usage:
n=1000
x0 = np.zeros(n)  ## initial guess
A_sparse, A_dense_v1, b =
generate_corrected_sparse_tridiagonal_matrix(n)
A_dense_v2 = A_sparse.toarray()  # Convert to dense format for
comparison




# Classical Jacobi (dense)
x_dense, iter_dense, time_dense = jacobi_dense(A_dense_v2, b, x0)

# Jacobi for sparse matrix
x_sparse, iter_sparse, time_sparse = jacobi_sparse(A_sparse, b,
x0)

print(f"Iterations (dense): {iter_dense}, Time (dense):
{time_dense:.4f} seconds")
print(f"Iterations (sparse): {iter_sparse}, Time (sparse):
{time_sparse:.4f} seconds")

#x_exact = np.linalg.solve(A_dense_v2, b)
#print(x_exact)
#print(x_sparse)
```

6) GS_Jacobisparse

```
import numpy as np
import scipy.sparse as sparse
from scipy.sparse import csr_matrix
import matplotlib.pyplot as plt
```

```python
def generate_simple_sparse_tridiagonal_matrix(n, diagonal_value=10,
off_diagonal_value=4):
    """
    Generates a sparse tridiagonal matrix, ensuring no overlaps.

    Args:
        n: Dimension of the system (size of the matrix A).
        diagonal_value: Value for the diagonal elements.
        off_diagonal_value: Value for the off-diagonal elements.

    Returns:
        A: Sparse coefficient matrix (scipy.sparse.csr_matrix).
        A_dense: equivalent Dense matrix (numpy array)
        b: Right-hand side vector (numpy array).
    """
    # Main diagonal
    main_diag = np.full(n, diagonal_value) #vecteur de taille (1,n) ne
contenant que des 5 (valeur de la diagonale)
    off_diag = np.full(n-1,off_diagonal_value) #vecteur de taille
(1,n-1) ne contenant que des 1 (valeur de la sur-diagonale et de la
sous-diagonale)

    # Construct sparse matrix
    data = np.concatenate([main_diag, off_diag, off_diag])
    rows = np.concatenate([np.arange(n), np.arange(n-1),
np.arange(1,n)])
    cols = np.concatenate([np.arange(n), np.arange(1,n),
np.arange(n-1)])
    As = csr_matrix((data, (rows, cols)), shape=(n, n))
    #print(As)

    # Construct dense matrix for reference
    A_dense = np.zeros((n, n))
    A_dense[n-1,n-1]=diagonal_value
    for i in range(n-1):
        A_dense[i, i] = diagonal_value
        A_dense[i, i+1]= off_diagonal_value
        A_dense[i+1,i]=off_diagonal_value

    b = np.random.rand(n)

    return As, A_dense, b
```

```python
def generate_sparse_tridiagonal_matrix(n):
    """
    Generates a sparse tridiagonal matrix with the specific values.

    Args:
        n: Dimension of the system (size of the matrix A).

    Returns:
        A: Sparse coefficient matrix (scipy.sparse.csr_matrix).
        b: Right-hand side vector (numpy array).
    """
    # Main diagonal
    diagonal_value = 2
    off_diagonal_value = -1
    main_diag = np.full(n, diagonal_value) #vecteur de taille (1,n) ne contenant que des 5 (valeur de la diagonale)
    off_diag = np.full(n-1,off_diagonal_value) #vecteur de taille (1,n-1) ne contenant que des 1 (valeur de la sur-diagonale et de la sous-diagonale)

    # Construct sparse matrix
    h=1/(n+1)
    data = np.concatenate([main_diag, off_diag, off_diag])
    rows = np.concatenate([np.arange(n), np.arange(n-1), np.arange(1,n)])
    cols = np.concatenate([np.arange(n), np.arange(1,n), np.arange(n-1)])
    As = (1/(h**2)) * csr_matrix((data, (rows, cols)), shape=(n, n))
    #print(As)

    # Construct dense matrix for reference
    A_dense = np.zeros((n, n))
    A_dense[n-1,n-1]=diagonal_value
    for i in range(n-1):
        A_dense[i, i] = diagonal_value
        A_dense[i, i+1]= off_diagonal_value
        A_dense[i+1,i]=off_diagonal_value
    A_dense = (1/(h**2)) * A_dense
    b = np.random.rand(n)

    return As,  A_dense, b
```

```python
def jacobi_sparse_with_error(A, b, x0, x_exact, tol=1e-5,
max_iter=1000):
    """
    Jacobi method for sparse matrices.

    Args:
        A: Sparse coefficient matrix (scipy.sparse.csr_matrix).
        b: Right-hand side vector (numpy array).
        x0: Initial guess for the solution vector (numpy array).
        x_exact: The true solution, used to compute ||x^(k) - x_exact||
        tol: Tolerance for convergence.
        max_iter: Maximum number of iterations.

    Returns:
        x: The found solution.
        iterations: Number of iterations performed.
        errors: List of errors between exact and approximate solution
for each iteration.
    """



    x = x0.copy()
    errors = []
    D1=1/A.diagonal()
    LU=A-sparse.diags(A.diagonal()) #correspond à -(L+U)

    for k in range(max_iter):
        x_new=D1*(b-LU.dot(x))

        error = np.linalg.norm(x_new - x_exact)
        errors.append(error)

        x = x_new
        if error < tol:
            break

    return x, k+1, errors


def gauss_seidel_sparse_with_error(A, b, x0, x_exact, tol=1e-5,
max_iter=1000):
    """
```

```
    Jacobi method for sparse matrices.

    Args:
        A: Sparse coefficient matrix (scipy.sparse.csr_matrix).
        b: Right-hand side vector (numpy array).
        x0: Initial guess for the solution vector (numpy array).
        x_exact: The true solution, used to compute ||x^(k) - x_exact||
        tol: Tolerance for convergence.
        max_iter: Maximum number of iterations.

    Returns:
        x: The found solution.
        iterations: Number of iterations performed.
        errors: List of errors between exact and approximate solution
for each iteration.
    """

    x=x0.copy()
    errors = []
    DL=sparse.tril(A) #correspond à D-L selon la formule du cours
    U=sparse.triu(A) - sparse.diags(A.diagonal()) #correspond à -U
    DL_inv = sparse.linalg.inv(DL)

    for k in range(max_iter):
        x_new=DL_inv * (b - U.dot(x))
        error = np.linalg.norm(x_new - x_exact)
        errors.append(error)

        if error < tol:
            break
        x = x_new

    return x, k+1, errors

def plot_error(errors, iterations, name):
    plt.figure(figsize=(8, 6))
    plt.plot(range(iterations), errors, marker='o', linestyle='-')
    plt.semilogy(range(iterations), errors, marker='o', linestyle='-')
# Use semilogy for log-scale on y-axis
    plt.xlabel("Iterations")
    plt.ylabel("Error estimate")
    plt.title("Error vs Iterations" + name)
    plt.grid(True)
```

```
    plt.show()

n=100
x0 = np.zeros(n)
As1, A_dense1, b1 = generate_simple_sparse_tridiagonal_matrix(n)
As2, A_dense2, b2 =generate_sparse_tridiagonal_matrix(n)

x_exact1 = sparse.linalg.inv(As1) * b1
x_exact2 = sparse.linalg.inv(As2) * b2

# Solve using Jacobi sparse method with error
x_jacobi1, iter_jacobi1, errors_jacobi1 = jacobi_sparse_with_error(As1,
b1, x0, x_exact1)
x_jacobi2, iter_jacobi2, errors_jacobi2 = jacobi_sparse_with_error(As2,
b2, x0, x_exact2)

# Solve using Gauss Seidel sparse with error
x_gs1, iter_gs1, errors_gs1 = gauss_seidel_sparse_with_error(As1, b1,
x0, x_exact1)
x_gs2, iter_gs2, errors_gs2 = gauss_seidel_sparse_with_error(As2, b2,
x0, x_exact2)

## Print results
#print(f"On génère A avec generate_simple_sparse_tridiagonal_matrix :")
#print(f"Avec Jacobi on trouve : x={x_jacobi1}, {iter_jacobi1}
iterations, erreurs:{errors_jacobi1}")
#print(f"Avec Gauss Seidel on trouve : x={x_gs1}, {iter_gs1}
iterations, erreurs:{errors_gs1}")

#print(f"On génère A avec generate_sparse_tridiagonal_matrix :")
#print(f"Avec Jacobi on trouve : x={x_jacobi2}, {iter_jacobi2}
iterations, erreurs:{errors_jacobi2}")
#print(f"Avec Gauss Seidel on trouve : x={x_gs2}, {iter_gs2}
iterations, erreurs:{errors_gs2}")


plot_error(errors_jacobi1, iter_jacobi1, " Jacobi Sparse 1")
plot_error(errors_jacobi2, iter_jacobi2, " Jacobi Sparse 2")
plot_error(errors_gs1, iter_gs1, " Gauss Siedel 1")
plot_error(errors_gs2, iter_gs2, " Gauss Siedel 2")
```

7) sparse_gs_comparaison

```python
import numpy as np
import scipy.sparse as sparse
from scipy.sparse import csr_matrix
import matplotlib.pyplot as plt

def generate_simple_sparse_tridiagonal_matrix(n, diagonal_value=10,
off_diagonal_value=4):
    """
    Generates a sparse tridiagonal matrix, ensuring no overlaps.

    Args:
        n: Dimension of the system (size of the matrix A).
        diagonal_value: Value for the diagonal elements.
        off_diagonal_value: Value for the off-diagonal elements.

    Returns:
        A: Sparse coefficient matrix (scipy.sparse.csr_matrix).
        A_dense: equivalent Dense matrix (numpy array)
        b: Right-hand side vector (numpy array).
    """
    # Main diagonal
    main_diag = np.full(n, diagonal_value) #vecteur de taille (1,n) ne
contenant que des 5 (valeur de la diagonale)
    off_diag = np.full(n-1,off_diagonal_value) #vecteur de taille
(1,n-1) ne contenant que des 1 (valeur de la sur-diagonale et de la
sous-diagonale)

    # Construct sparse matrix
    data = np.concatenate([main_diag, off_diag, off_diag])
    rows = np.concatenate([np.arange(n), np.arange(n-1),
np.arange(1,n)])
    cols = np.concatenate([np.arange(n), np.arange(1,n),
np.arange(n-1)])
    As = csr_matrix((data, (rows, cols)), shape=(n, n))
    #print(As)

    # Construct dense matrix for reference
    A_dense = np.zeros((n, n))
    A_dense[n-1,n-1]=diagonal_value
    for i in range(n-1):
        A_dense[i, i] = diagonal_value
        A_dense[i, i+1]= off_diagonal_value
        A_dense[i+1,i]=off_diagonal_value
```

```
    b = np.random.rand(n)

    return As, A_dense, b

def generate_sparse_tridiagonal_matrix(n):
    """
    Generates a sparse tridiagonal matrix with the specific values.

    Args:
        n: Dimension of the system (size of the matrix A).

    Returns:
        A: Sparse coefficient matrix (scipy.sparse.csr_matrix).
        b: Right-hand side vector (numpy array).
    """
    # Main diagonal
    diagonal_value = 2
    off_diagonal_value = -1
    main_diag = np.full(n, diagonal_value) #vecteur de taille (1,n) ne
contenant que des 5 (valeur de la diagonale)
    off_diag = np.full(n-1,off_diagonal_value) #vecteur de taille
(1,n-1) ne contenant que des 1 (valeur de la sur-diagonale et de la
sous-diagonale)

    # Construct sparse matrix
    h=1/(n+1)
    data = np.concatenate([main_diag, off_diag, off_diag])
    rows = np.concatenate([np.arange(n), np.arange(n-1),
np.arange(1,n)])
    cols = np.concatenate([np.arange(n), np.arange(1,n),
np.arange(n-1)])
    As = (1/(h**2)) * csr_matrix((data, (rows, cols)), shape=(n, n))
    #print(As)

    # Construct dense matrix for reference
    A_dense = np.zeros((n, n))
    A_dense[n-1,n-1]=diagonal_value
    for i in range(n-1):
        A_dense[i, i] = diagonal_value
        A_dense[i, i+1]= off_diagonal_value
        A_dense[i+1,i]=off_diagonal_value
    A_dense = (1/(h**2)) * A_dense
```

```
    b = np.random.rand(n)

    return As,  A_dense, b



def jacobi_sparse_with_error(A, b, x0, x_exact, tol=1e-5,
max_iter=1000):
    """
    Jacobi method for sparse matrices.

    Args:
        A: Sparse coefficient matrix (scipy.sparse.csr_matrix).
        b: Right-hand side vector (numpy array).
        x0: Initial guess for the solution vector (numpy array).
        x_exact: The true solution, used to compute ||x^(k) - x_exact||
        tol: Tolerance for convergence.
        max_iter: Maximum number of iterations.

    Returns:
        x: The found solution.
        iterations: Number of iterations performed.
        errors: List of errors between exact and approximate solution
for each iteration.
    """



    x = x0.copy()
    errors = []
    D1=1/A.diagonal()
    LU=A-sparse.diags(A.diagonal()) #correspond à -(L+U)

    for k in range(max_iter):
        x_new=D1*(b-LU.dot(x))

        error = np.linalg.norm(x_new - x_exact)
        errors.append(error)

        x = x_new
        if error < tol:
            break

    return x, k+1, errors
```

```python
def gauss_seidel_sparse_with_error(A, b, x0, x_exact, tol=1e-5,
max_iter=1000):
    """
    Jacobi method for sparse matrices.

    Args:
        A: Sparse coefficient matrix (scipy.sparse.csr_matrix).
        b: Right-hand side vector (numpy array).
        x0: Initial guess for the solution vector (numpy array).
        x_exact: The true solution, used to compute ||x^(k) - x_exact||
        tol: Tolerance for convergence.
        max_iter: Maximum number of iterations.

    Returns:
        x: The found solution.
        iterations: Number of iterations performed.
        errors: List of errors between exact and approximate solution
for each iteration.
    """

    x=x0.copy()
    errors = []
    DL=sparse.tril(A) #correspond à D-L selon la formule du cours
    U=sparse.triu(A) - sparse.diags(A.diagonal()) #correspond à -U
    DL_inv = sparse.linalg.inv(DL)

    for k in range(max_iter):
        x_new=DL_inv * (b - U.dot(x))
        error = np.linalg.norm(x_new - x_exact)
        errors.append(error)

        if error < tol:
            break
        x = x_new

    return x, k+1, errors

def plot_error(errors, iterations, errors2, iterations2, name):
    plt.figure(figsize=(8, 6))
    plt.semilogy(range(iterations), errors, marker='o', linestyle='-',
label='Jacobi Sparse')  # Use semilogy for log-scale on y-axis
```

```
    plt.semilogy(range(iterations2), errors2, marker='o',
linestyle='-', label='Gauss Siedel')  # Use semilogy for log-scale on
y-axis
    plt.xlabel("Iterations")
    plt.ylabel("Error estimate")
    plt.title("Error vs Iterations" + name)
    plt.grid(True)
    plt.show()

n=100
x0 = np.zeros(n)
As1, A_dense1, b1 = generate_simple_sparse_tridiagonal_matrix(n)
As2, A_dense2, b2 =generate_sparse_tridiagonal_matrix(n)

x_exact1 = sparse.linalg.inv(As1) * b1
x_exact2 = sparse.linalg.inv(As2) * b2

# Solve using Jacobi sparse method with error
x_jacobi1, iter_jacobi1, errors_jacobi1 = jacobi_sparse_with_error(As1,
b1, x0, x_exact1)
x_jacobi2, iter_jacobi2, errors_jacobi2 = jacobi_sparse_with_error(As2,
b2, x0, x_exact2)

# Solve using Gauss Seidel sparse with error
x_gs1, iter_gs1, errors_gs1 = gauss_seidel_sparse_with_error(As1, b1,
x0, x_exact1)
x_gs2, iter_gs2, errors_gs2 = gauss_seidel_sparse_with_error(As2, b2,
x0, x_exact2)

plot_error(errors_jacobi1, iter_jacobi1, errors_gs1, iter_gs1, " Jacobi
Sparse 1 VS Gauss Siedel 1")
plot_error(errors_jacobi2, iter_jacobi2, errors_gs2, iter_gs2, " Jacobi
Sparse 1 VS Gauss Siedel 2")
```

8) SOR

```
import numpy as np
import scipy.sparse as sparse
from scipy.sparse import csr_matrix
import matplotlib.pyplot as plt
```

```
def generate_simple_sparse_tridiagonal_matrix(n, diagonal_value=10,
off_diagonal_value=4):
    main_diag = np.full(n, diagonal_value)
    off_diag = np.full(n-1, off_diagonal_value)

    # Construct sparse matrix
    data = np.concatenate([main_diag,off_diag,off_diag])
    rows = np.concatenate([np.arange(n), np.arange(n-1),
np.arange(1,n)])
    cols = np.concatenate([np.arange(n), np.arange(1,n),
np.arange(n-1)])
    As = csr_matrix((data, (rows, cols)), shape=(n, n))

    # Construct dense matrix for reference
    A_dense = np.zeros((n, n))
    A_dense[n-1,n-1]=diagonal_value
    for i in range(n-1):
        A_dense[i, i] = diagonal_value
        A_dense[i,i+1]=off_diagonal_value
        A_dense[i+1,i]=off_diagonal_value
    b = np.random.rand(n)
    return As, A_dense, b

def successive_over_relaxation(A,b,x0, x_exact, tol=1e-5,
max_iter=1000,w=1):
    """
    Args:
        A: Sparse coefficient matrix (scipy.sparse.csr_matrix).
        b: Right-hand side vector (numpy array).
        x0: Initial guess for the solution vector (numpy array).
        x_exact : The true solution, used to compute ||x^(k)-x_exact||
        tol: Tolerance for convergence.
        max_iter: Maximum number of iterations.
        w: relaxiation parameter (0<w<2)
    Returns:
        x : Approximate solution vector.
        iterations: Number of iterations performed.
        errors: List of errors between exact and approximate solution
at each iteration.
    """
    x=x0.copy()
    D=sparse.diags(A.diagonal())
    L=sparse.tril(A,k=-1)
```

```
    U=sparse.triu(A,k=1)
    C=(1/w)*(D+w*L)
    errors = []
    C1=sparse.linalg.inv(C)
    for k in range(max_iter):
        x_new = C1.dot(b-(((w-1)/w)*D+U).dot(x))
        error = np.linalg.norm(x_new-x_exact)
        errors.append(error)
        if error < tol:
            break
        x = x_new
    return x, k + 1, errors

def plot_error(errors, iterations):
    plt.figure(figsize=(8, 6))
    plt.plot(range(iterations), errors, marker='o', linestyle='-')
    plt.semilogy(range(iterations), errors, marker='o', linestyle='-')
# Use semilogy for log-scale on y-axis
    plt.xlabel("Iterations")
    plt.ylabel("Error estimate")
    plt.title("Error vs Iterations (SOR)")
    plt.grid(True)
    plt.show()

n=100
x0=np.zeros(n)
A, A_dense, b=generate_simple_sparse_tridiagonal_matrix(n,
diagonal_value=10, off_diagonal_value=4)
x_exact=sparse.linalg.inv(A)*b
x, iter, errors=successive_over_relaxation(A,b,x0, x_exact)

print(f"Avec SOR on trouve : x={x}, {iter} iterations,
erreurs:{errors}")
plot_error(errors, iter)
```

9) best_omega

```
import numpy as np
import scipy.sparse as sparse
from scipy.sparse import csr_matrix
import matplotlib.pyplot as plt
```

```
def generate_simple_sparse_tridiagonal_matrix(n, diagonal_value=10,
off_diagonal_value=4):
    main_diag = np.full(n, diagonal_value)
    off_diag = np.full(n-1, off_diagonal_value)

    # Construct sparse matrix
    data = np.concatenate([main_diag,off_diag,off_diag])
    rows = np.concatenate([np.arange(n), np.arange(n-1),
np.arange(1,n)])
    cols = np.concatenate([np.arange(n), np.arange(1,n),
np.arange(n-1)])
    As = csr_matrix((data, (rows, cols)), shape=(n, n))

    # Construct dense matrix for reference
    A_dense = np.zeros((n, n))
    A_dense[n-1,n-1]=diagonal_value
    for i in range(n-1):
        A_dense[i, i] = diagonal_value
        A_dense[i,i+1]=off_diagonal_value
        A_dense[i+1,i]=off_diagonal_value
    b = np.random.rand(n)
    return As, A_dense, b

def Sucessive_Over_Relaxation(A, b, w, x0, x_exact, tol=1e-5,
max_iter=100):
    """
    SOR method for sparse matrices.

    Args:
        A: Sparse coefficient matrix (scipy.sparse.csr_matrix).
        b: Right-hand side vector (numpy array).
        w: relaxiation parameter (0<w<2)
        x0: Initial guess for the solution vector (numpy array).
        x_exact: The true solution, used to compute ||x^(k) - x_exact||
        tol: Tolerance for convergence.
        max_iter: Maximum number of iterations.

    Returns:
        x: The found solution.
        iterations: Number of iterations performed.
        errors: List of errors between exact and approximate solution
for each iteration.
    """
```

```
    x=x0.copy()
    errors = []
    D=sparse.diags(A.diagonal())
    L=sparse.tril(A, k=-1) #correspond à -L
    U=sparse.triu(A, k=1) #correspond à -U
    C=(1/w)*(D+(w*L))
    C_inv=sparse.linalg.inv(C)

    for k in range(max_iter):
        x_new = C_inv.dot(b - (((w-1)/w)*D + U).dot(x))
        error = np.linalg.norm(x_new - x_exact)
        errors.append(error)

        if error < tol:
            break
        x = x_new

    return x, k+1, errors


def best_omega(As, b,x0,x_exact, tol=1e-5, max_iter=100):
    min_iter=max_iter
    best_w=0
    val=np.arange(0.1,2.0,0.1)
    for w in val:
        x, iter, errors = Sucessive_Over_Relaxation(As, b, w, x0,
x_exact, tol=tol, max_iter=max_iter)
        if (iter<min_iter):
            min_iter=iter
            best_w=w

    return best_w

def plot_error_omegas(As, b, x0, x_exact, tol=1e-5, max_iter=100):
    omegas = np.arange(0.1, 2.0, 0.1)
    plt.figure(figsize=(12, 8))

    for w in omegas:
        x, iter, errors = Sucessive_Over_Relaxation(As, b, w, x0,
x_exact, tol=tol, max_iter=max_iter)
        plt.semilogy(range(len(errors)), errors, marker='o',
linestyle='-', label='ω = ' + str(round(w, 1)))
```

```
    plt.title("Error vs Iteration for Different Omega Values")
    plt.xlabel("Iteration")
    plt.ylabel("Error")
    plt.legend()
    plt.grid(True)
    plt.show()

n=100
As, A_dense, b =generate_simple_sparse_tridiagonal_matrix(n,
diagonal_value=10, off_diagonal_value=4)
x0 = np.zeros(n)
x_exact = sparse.linalg.spsolve(As,b)
meilleur=best_omega(As, b,x0,x_exact)
print(f"Meilleur Omega {meilleur}")
plot_error_omegas(As, b, x0, x_exact)
```

10) comparaison_A3_A4

```
import numpy as np
import matplotlib.pyplot as plt
import time

def jacobi_method(A, b, x0, tol=1e-5, max_iter=1000):
  """
  Implements the Jacobi method for solving the linear system Ax = b.

  Args:
    A: Coefficient matrix (numpy array).
    b: Right-hand side vector (numpy array).
    x0: Initial guess for the solution vector (numpy array).
    tol: Tolerance for convergence.
    max_iter: Maximum number of iterations.

  Returns:
    x: Approximate solution vector.
    iterations: Number of iterations performed.
    errors: List of errors between exact and approximate solution at
each iteration.
  """
  ### TODO: Review code here
  start_time=time.time()
  n = A.shape[0]
  x = x0.copy()
```

```
  errors = []
  for k in range(max_iter):
    x_new = np.zeros_like(x)
    for i in range(n):
        x_new[i]=(1/A[i,i])*(b[i]-np.dot(A[i,:],x)+A[i,i]*x[i])
    error = np.linalg.norm(x_new-x)
    errors.append(error)
    x = x_new
    if error < tol:
        break
  end_time=time.time()
  time_taken=end_time-start_time
  #calcul du rayon spectral
  D=np.diag(np.diag(A))
  T=np.matmul(np.linalg.inv(D),D-A)
  r=np.max(np.absolute(np.linalg.eigvals(T)))
  #matrice dominante
  if
np.all(((np.absolute(np.diag(A))))>np.sum(np.absolute(A-D),axis=1)):
#axis=1 colonne
    print('La matrice est dominante')
  else:
    print("La matrice n'est pas dominante")
  return x, k + 1, errors, time_taken, r

def gauss_seidel_dense_with_error(A, b, x0, x_exact, tol=1e-5,
max_iter=1000):
    ### TODO:
    """
    Args:
        A: Sparse coefficient matrix (scipy.sparse.csr_matrix).
        b: Right-hand side vector (numpy array).
        x0: Initial guess for the solution vector (numpy array).
        x_exact : The true solution, used to compute ||x^(k)-x_exact||
        tol: Tolerance for convergence.
        max_iter: Maximum number of iterations.
    Returns:
        x : Approximate solution vector.
        iterations: Number of iterations performed.
        errors: List of errors between exact and approximate solution
at each iteration.
    """
    start_time=time.time()
```

```
    x=x0.copy()
    C=np.tril(A)  #correspond à D-L
    U=np.triu(A)-np.diag(np.diag(A))
    errors = []
    C1=np.linalg.inv(C)
    for k in range(max_iter):
        x_new = C1.dot(b-np.dot(U,x))
        error = np.linalg.norm(x_new-x_exact)
        errors.append(error)
        if error < tol:
            break
        x = x_new
    end_time=time.time()
    time_taken=end_time-start_time
    return x, k + 1, errors, time_taken



def plot_error(errors, iterations, errors2, iterations2, name):
    plt.figure(figsize=(8, 6))
    plt.semilogy(range(iterations), errors, marker='o', linestyle='-')
# Use semilogy for log-scale on y-axis
    plt.semilogy(range(iterations2), errors2, marker='o',
linestyle='-')  # Use semilogy for log-scale on y-axis
    plt.xlabel("Iterations")
    plt.ylabel("Error estimate")
    plt.title("Error vs Iterations"+ name)
    plt.grid(True)
    plt.show()

# Example usage:
n = 100
A3=np.array([[4,1,1],[2,-9,0],[0,-8,-6]])
A4=np.array([[7,6,9],[4,5,-4],[-7,-3,8]])
b3=np.array([6,-7,-14])
b4=np.array([22,5,-2])
x0 = np.zeros(np.size(b3))
# Calculate exact solution
x_exact1 = np.linalg.solve(A3, b3)
x_exact2 = np.linalg.solve(A4, b4)

# Solve using Jacobi method
```

```
x_jacobi1, iterations_jacobi1, errors_jacobi1, time_jacobi1, r_1 =
jacobi_method(A3, b3, x0)
x_jacobi2, iterations_jacobi2, errors_jacobi2, time_jacobi2, r_2 =
jacobi_method(A4, b4, x0)
x_gs1, iterations_gs1, errors_gs1,
time_gs1=gauss_seidel_dense_with_error(A3, b3, x0, x_exact1)
x_gs2, iterations_gs2, errors_gs2,
time_gs2=gauss_seidel_dense_with_error(A4, b4, x0, x_exact2)

# Print results
print(f"Pour la matrice A3 (Jacobi), Solution Jacobi: {x_jacobi1},
Iterations: {iterations_jacobi1}, Temps: {time_jacobi1}") #temps
renvoyé 0.0003616809844970703
print(f"Pour la matrice A4 (Gauss Seidel), Solution GS: {x_gs1},
Iterations: {iterations_gs1}, Temps: {time_gs1}")
print(f"Pour la matrice A3, Rayon spectral: {r_1}, Exact solution:
{x_exact1}")
print(f"Pour la matrice A4 (Jacobi), Solution Jacobi: {x_jacobi2},
Iterations: {iterations_jacobi2}, Temps: {time_jacobi2}")#temps renvoyé
0.016640663146972656
print(f"Pour la matrice A4 (Gauss Seidel), Solution GS: {x_gs2},
Iterations: {iterations_gs2}, Temps: {time_gs2}")
print(f"Pour la matrice A4, Exact solution: {x_exact2}, Rayon spectral:
{r_2}")

# Plot the error
plot_error(errors_jacobi1, iterations_jacobi1, errors_gs1,
iterations_gs1, " Matrice A3 (Jacobi VS Gauss Seidel)")
plot_error(errors_jacobi2, iterations_jacobi2, errors_gs2,
iterations_gs2, " Matrice A4 (Jacobi VS Gauss Seidel)")
```