Système linéaire grande dimension

COSTANTIN Perline

ZIAD Zineb

Projet MAM3

Sommaire

Définition

- Une matrice carré A de taille n×n et un vecteur b de taille n×1, trouver un vecteur x de taille n × 1 qui satisfait :Ax = b
- Approche directe: Résoudre $x = A^{-1}b$

Jacobi Dense : Méthode

On considère le système linéaire Ax=b tel que A respecte :

$a_{ii} = 5(i+1)$ for $i = 1, 2, \ldots, n$ and $a_{ij} = -1$ for $i \neq j$, et b un vecteur aléatoire

4 méthodes :

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

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

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

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

Jacobi dense

Comparaison des temps d'itération selon les méthodes Jacobi dense

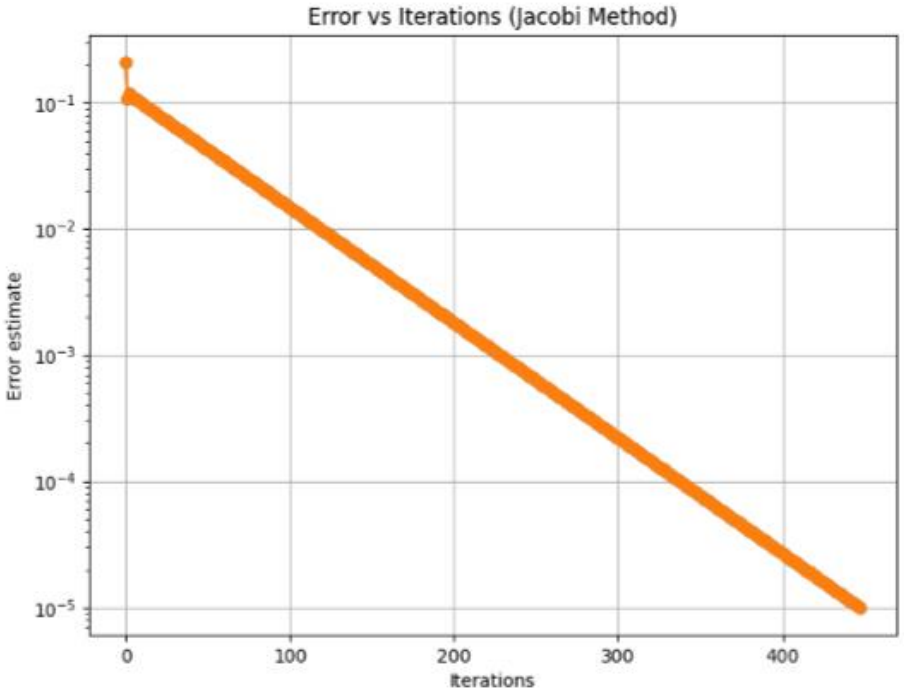
Temps d'itération pour n=	JD inverse	JD boucle	JD somme	JD dot
10	0,001059	0,000463	0.511357	0,229021
100	1,091545	0,981145	0.483927	0.208428
500	/	/	0,5004633	0,231787
JD inv et boucle ne marche pasà cause du rayon de cv				
Inv trop lent et trop de place				

Jacobi dense

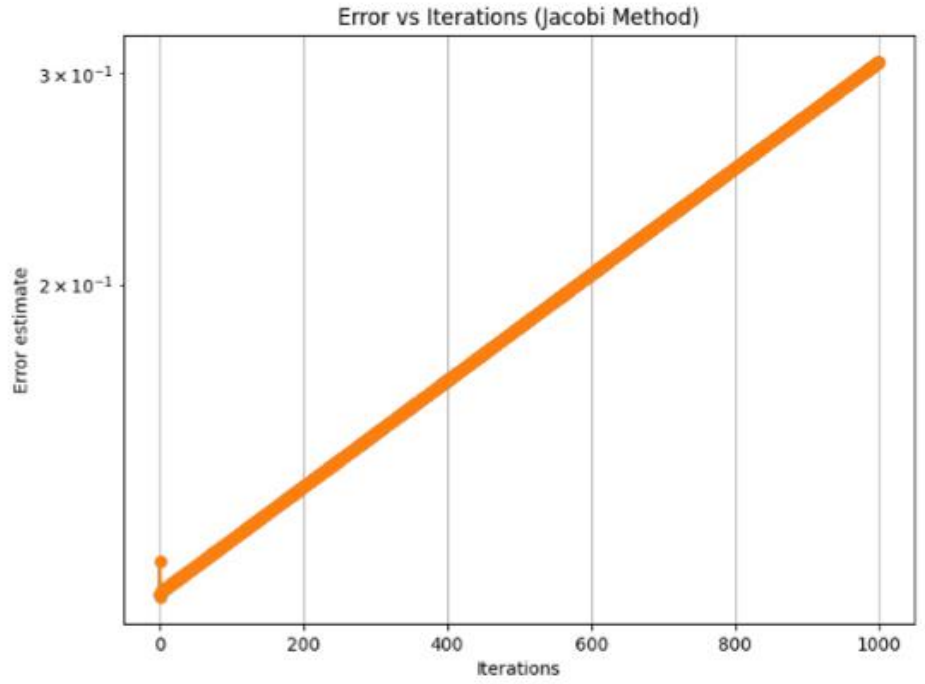
```python
def spectral_radius(A):
# Vérification du rayon spectral
  D = np.zeros_like(A)
  for i in range(n):
    D[i,i] = A[i,i]
  T = np.dot(np.linalg.inv(D),(A-D)) # Calcul de la matrice d'itération T
  rho = max(abs(np.linalg.eigvals(T)))  #Calcul du rayon spectral (valeur propre de plus grande norme de T)
  print("Le rayon spectral est égal à", rho)

  return rho

def diagonale_dominante(A):
# Vérifie si la matrice est de diagonale dominante
  n = A.shape[0]
  for i in range(n):
      # Somme des éléments hors-diagonale
      somme_hors_diag = sum(abs(A[i, j]) for j in range(n) if j != i)

      if (abs(A[i, i]) < somme_hors_diag):
          print("La matrice n'est pas diagonale dominante.")
          return

  print("La matrice est diagonale dominante.")
  return
```

Rayon Spectral

→ Code utilisé: Jacobi dense avec boucle
ρ=max(|λi|)ρ, λi les valeurs propres de T.
ρ≥1 => ne converge pas.



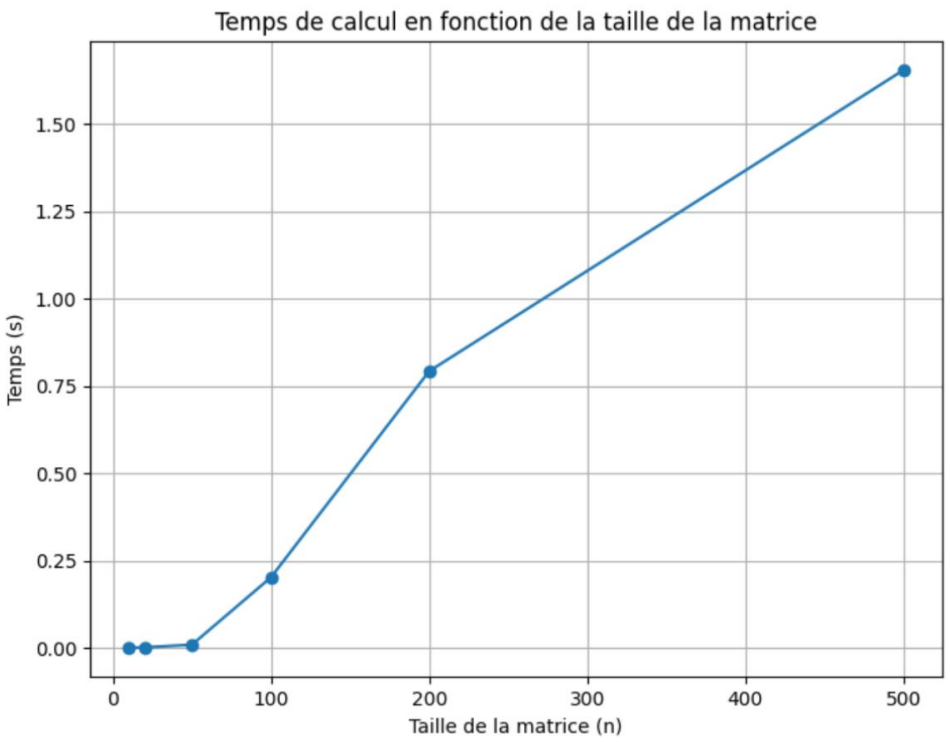
Pour n=100, on trouve ρ=0.97917067



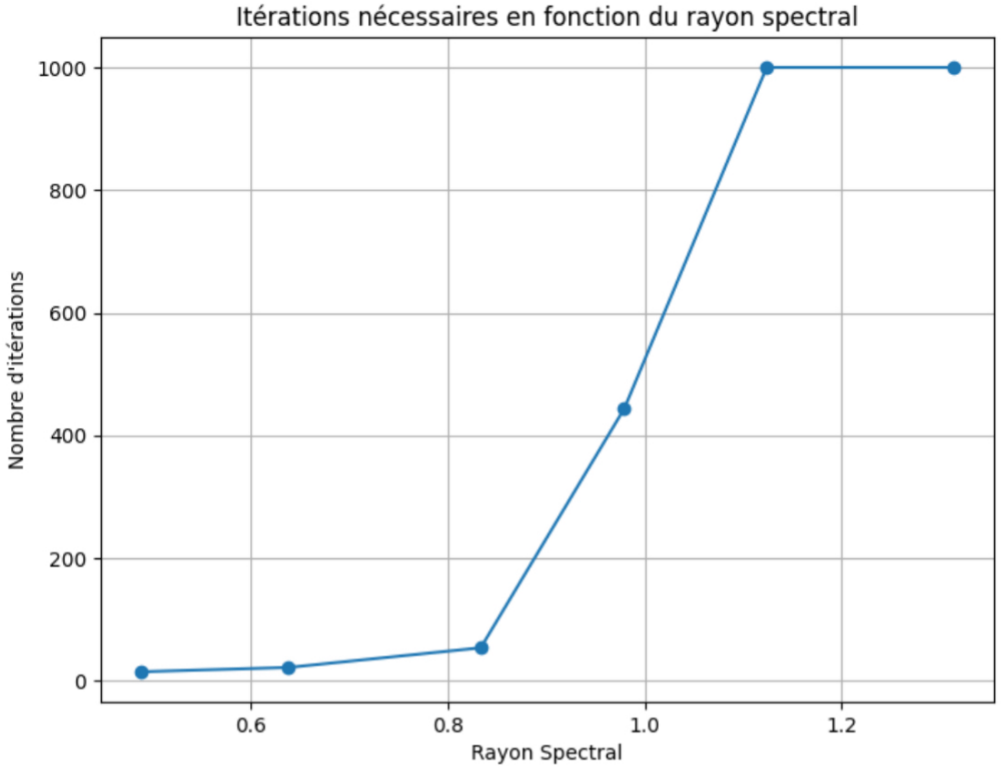
Pour n=111, on trouve ρ=1.0010234598

Diagonale dominante

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

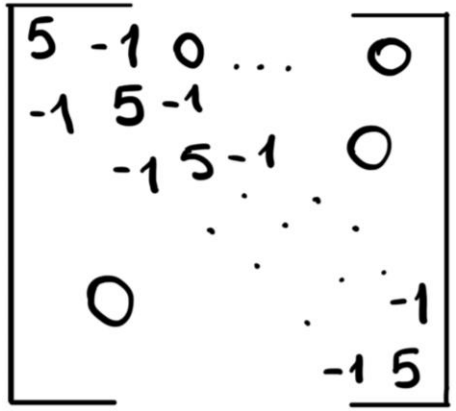


Rayon spectral (ρ) en fonction du nombre d'itérations



Jacobi Sparse

Une matrice Sparse est une matrice avec une forte proportion de 0



Format de Stockage : CSR, CSC, COO

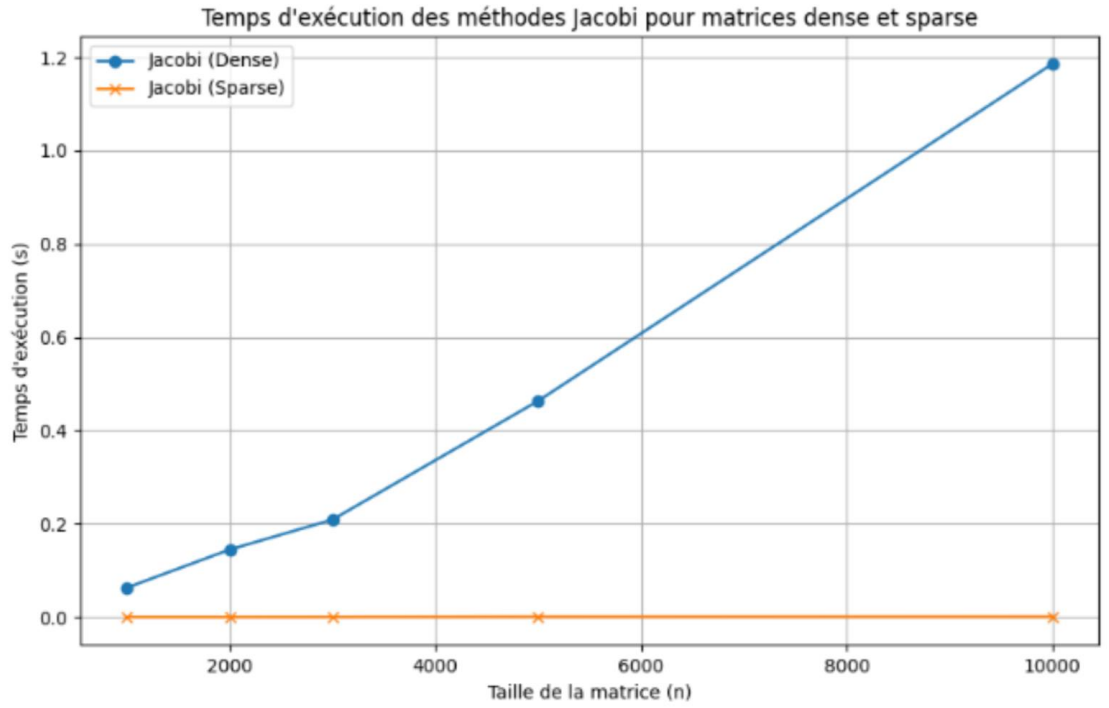
Jacobi Sparse

```python
def generate_corrected_sparse_tridiagonal_matrix(n, diagonal_value=5, off_diagonal_value=-1):
    main_diag = np.full(n, diagonal_value) #diagonale principale
    upp_diag= np.full(n-1,off_diagonal_value) #diagonale supérieure
    low_diag=np.full(n-1,off_diagonal_value)#diagonale inférieure
    # Construct sparse matrix
    data=np.concatenate((main_diag, upp_diag,low_diag ))
    rows = np.concatenate(( np.arange(n),np.arange(n-1),np.arange(1,n))) # might need to use np.concatenate
    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))
    for i in range(n):
        for j in range(n):
            if (np.abs(j - i) ==1) :
                A_dense[i,j] = off_diagonal_value
            elif i==j:
                A_dense[i, i] = diagonal_value
    b = np.random.rand(n)
    return As, A_dense, b
```

```python
def jacobi_sparse(A, b, x0, tol=1e-6, max_iter=10000):
    x = x0.copy()
    D_inv=1/A.diagonal()
    LU=A-sparse.diags(A.diagonal()) #-(L+U)
    start_time = time.time()
    for i in range(max_iter):
        x_new=D_inv*(b-LU.dot(x))
        error = np.linalg.norm(x_new - x)
        if error < tol:
            break
        x = x_new
    end_time = time.time()
    time_taken = end_time - start_time
    return x_new, i+1, time_taken
```

Jacobi sparse

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



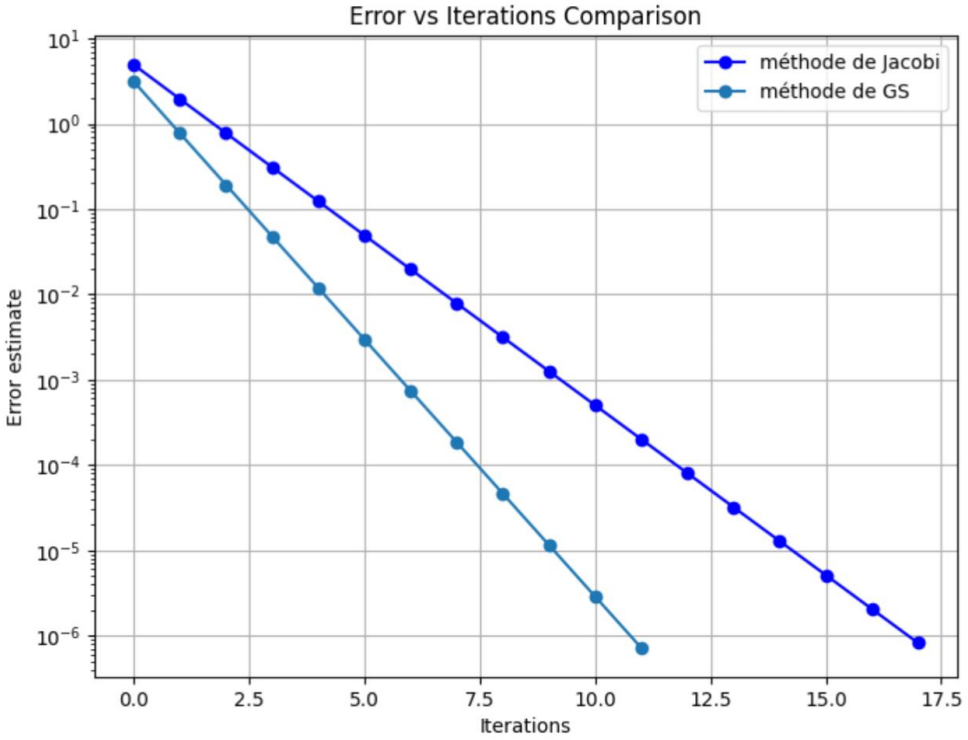
Gauss-Seidel

Gauss-Seidel: $C = D - L$ or $C = D - U$

Appliquer au système linéaire : $x = (D - L)^{-1}b + (D - L)^{-1}Ux$

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

Gauss Seidel



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

SOR

Ajout d'un paramètre de relaxation ω $(0 < \omega < 2)$

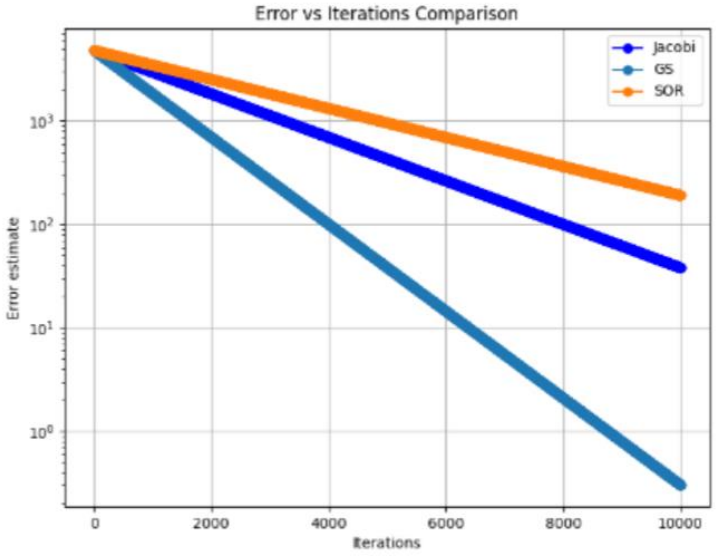
$$C = \frac{1}{\omega}(D - \omega L)$$

Formule itérative

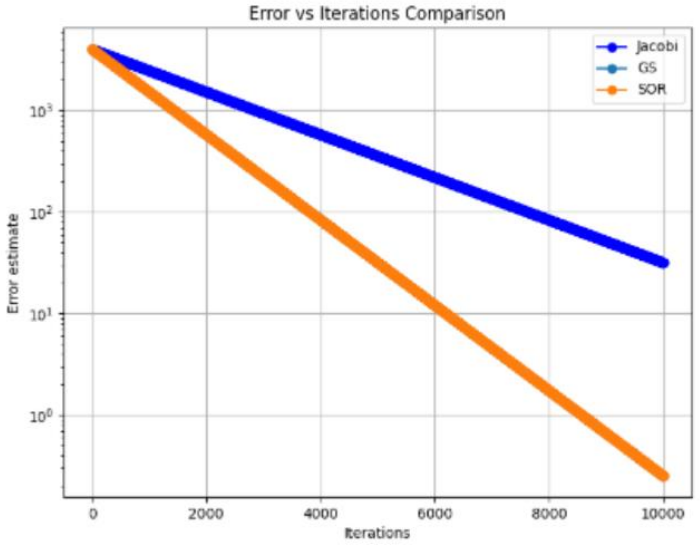
$$s_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right)$$

$$x_i^{(k+1)} = \omega s_i^{(k)} + (1 - \omega) x_i^{(k)}$$

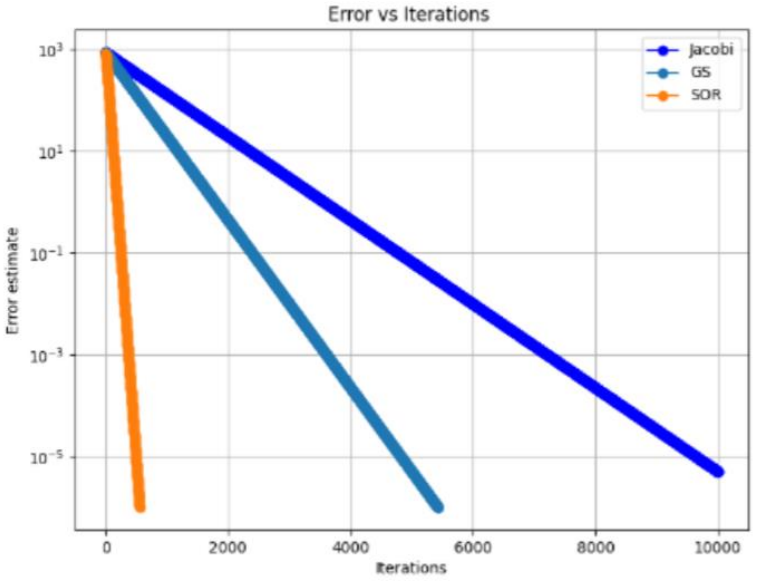
SOR



Pour ω =0.5



Pour ω =1



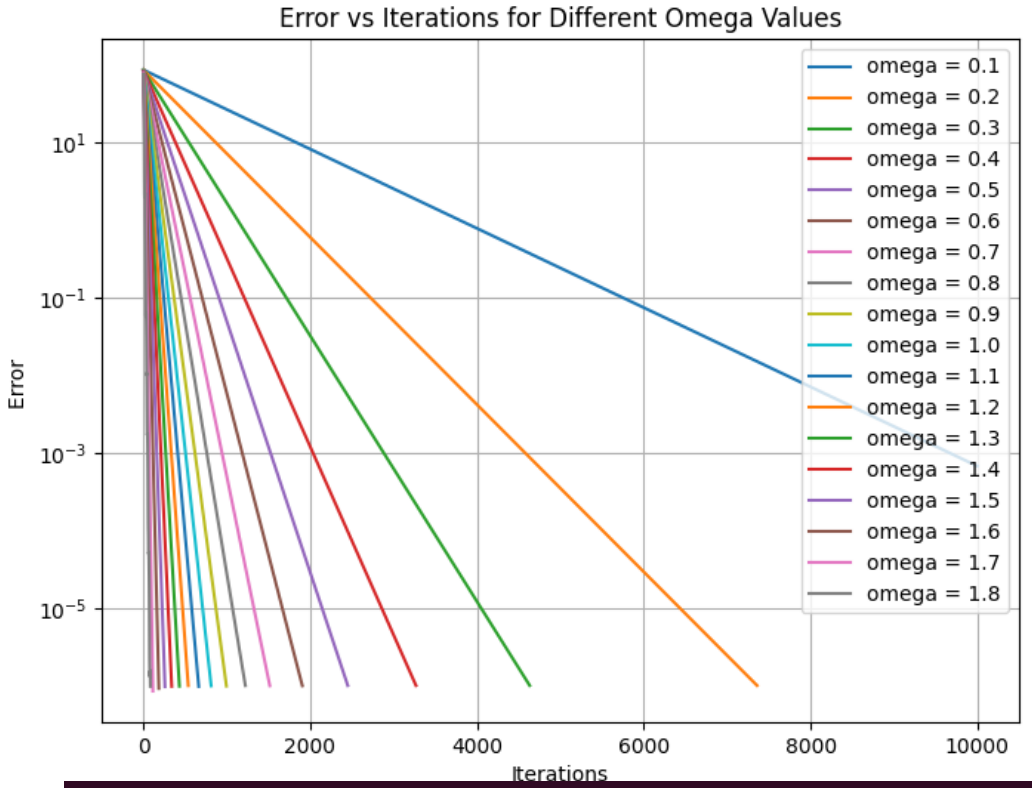
Pour ω =1.8

Tests Oméga

```python
def best_omega(A, b, x0, x_exact, tol=1e-6, max_iter=10000):
    best_w=0
    min_iter = max_iter
    val=np.arange(0.1,1.9,0.1)
    for omega in val:
        x, iterations, errors, time_taken = SOR(A, b, x0, x_exact, tol=tol, max_iter=max_iter, omega=omega)
        if (iterations<min_iter):
            min_iter=iterations
            best_w=omega
            best_time=time_taken
    print(f"Best omega: {best_w}, Iterations {min_iter}, Temps écoulé:{best_time:.4f} seconds")
    return best_w, min_iter,best_time

```

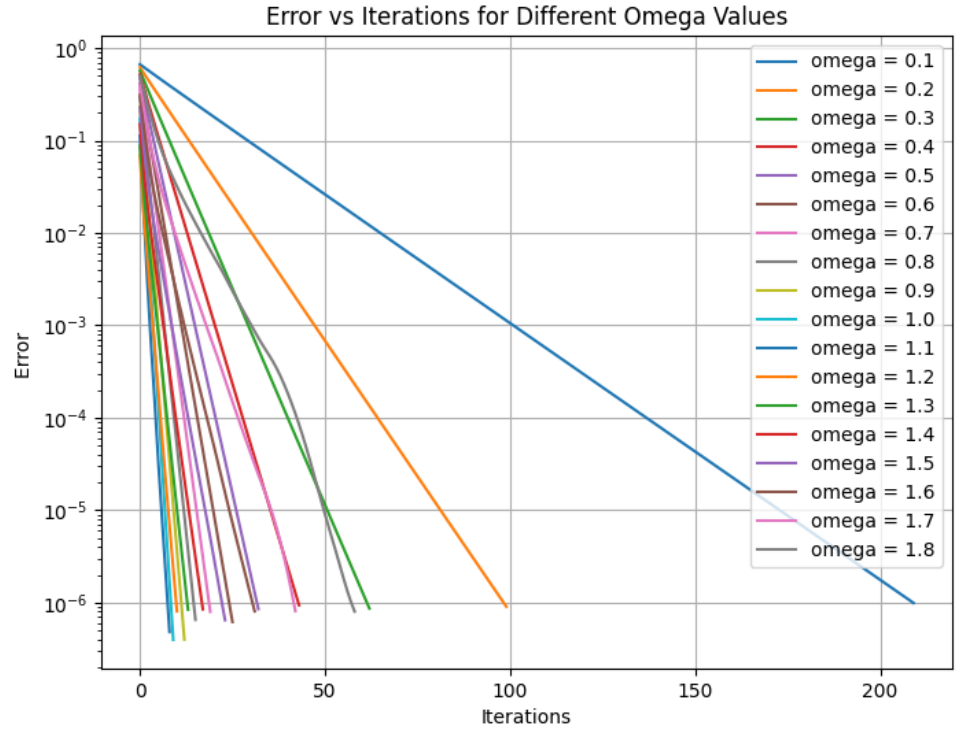
Tests Oméga (n=20)

matrice avec des 2 sur la diagonale et des -1 sur la sous-diagonale et la sur-diagonale



Best omega: 1.8000000000000003, Iterations 85,

matrice avec des 5 sur la diagonale et des -1 sur la sous-diagonale et la sur-diagonale



Best omega: 1.1, Iterations 9,

Merci pour votre attention