El ghalbzouri Akram TP2

October 25, 2023

1 Compte rendu TP2

2 Introduction

Ce travail pratique se concentre sur l'implémentation et les tests des méthodes de Jacobi et de Gauss-Seidel pour résoudre l'équation linéaire Ax = b, où A est une matrice et b est un vecteur. De plus, nous explorerons une méthode de relaxation visant à améliorer ces deux approches. La résolution de systèmes linéaires est essentielle dans divers domaines, et ces méthodes classiques sont cruciales pour converger vers des solutions précises.

3 Importation des bibliothèques nécessaires

Au début de notre travail, nous procédons à l'importation des bibliothèques essentielles qui seront utilisées pour l'implémentation des fonctions. Pour mener à bien nos tâches, nous dépendons de deux bibliothèques clés :

- **NumPy** : Cette bibliothèque est utilisée pour faciliter la manipulation de matrices, ce qui est essentiel pour les calculs liés à notre travail.
- **Math** : La bibliothèque `math` offre une gamme de fonctions mathématiques standard, qui seront utiles dans divers aspects de notre implémentation.

[ ]:
```
import numpy as np
import math
```

4 Jacobi

4.1 Méthode de Jacobi

Nous commençons par l'implémentation et l'explication de la méthode de Jacobi. Cette méthode est une technique itérative fondamentale utilisée pour résoudre des systèmes linéaires. Elle itère sur les composants du vecteur solution, mettant à jour chaque composant en fonction des valeurs précédemment calculées, tout en maintenant la stabilité de l'algorithme.

Nous allons décrire en détail les étapes de la méthode de Jacobi et fournir son implémentation pour résoudre efficacement des systèmes d'équations linéaires.

4.2 Fonction de calcul de la norme (Ordre 2)

Dans cette section, nous allons commencer par l'implémentation d'une fonction essentielle : le calcul de la norme de l'ordre 2 d'un vecteur. Cette fonction sera utilisée ultérieurement dans notre fonction principale pour effectuer des calculs essentiels.

[ ]:
```python
def norme(x) :
  norm = 0
  n = len(x)
  for i in range(n):
    norm += x[i] ** 2
  # print(norm)
  # if norm < 0 :
  #   norm = -norm
  return math.sqrt(norm)
```

4.3 Implementation du fonction :

[ ]:
```python
def jacobi(A, b, x0, N, eps):
    """
    Résout un système d'équations linéaires Ax = b en utilisant la méthode␣
 ↪itérative de Jacobi.

    Args:
        A (numpy.ndarray): Matrice des coefficients du système linéaire.
        b (numpy.ndarray): Vecteur des termes constants du système linéaire.
        x0 (numpy.ndarray): Vecteur initial de l'approximation de la solution.
        N (int): Nombre maximum d'itérations.
        eps (float): Tolérance pour le critère d'arrêt basé sur la norme.

    Returns:
        tuple: Un tuple contenant :
            - x (numpy.ndarray): La solution approximative du système.
            - iterations (int): Le nombre d'itérations effectuées.

    Si la méthode ne converge pas, la fonction renvoie (0, 0).
    """
    n = len(A)  # Nombre de lignes/colonnes de la matrice A
    nrm = 0  # Initialisation de la norme
    for k in range(N):  # Itération jusqu'à atteindre le nombre maximum␣
 ↪d'itérations (N)
        x = np.zeros(n)  # Initialisation d'un vecteur solution x

        # Boucle sur chaque élément du vecteur x
        for i in range(n):
            S = 0  # Initialisation d'une somme S
```

```
            # Boucle pour calculer la somme S
            for j in range(n):
                if j != i:
                    S += A[i, j] * x0[j]

            # Calcul de la nouvelle valeur de x[i] en utilisant la formule de␣
↪Jacobi
            x[i] = (b[i] - S) / A[i, i]

        # Test d'arrêt : calcul de la norme du vecteur différence entre x et x0
        nrm = norme(x - x0) ** 2 / (norme(x) ** 2)

        if nrm <= eps:
            return x, k + 1  # Retourne la solution x et le nombre d'itérations␣
↪nécessaires

        else:
            x0 = x  # Met à jour x0 avec la nouvelle valeur de x pour la␣
↪prochaine itération

    print("La méthode de Jacobi ne converge pas")
    return 0, 0  # Retourne 0, 0 en cas de non-convergence
```

5 Gaussiedel

Nous passons maintenant à l'implémentation et à l'explication de la méthode de Gauss-Seidel. Cette méthode est similaire à celle de Jacobi, mais elle utilise les nouvelles valeurs de x dès qu'elles sont calculées, ce qui peut améliorer la convergence.

Nous allons décrire les étapes de la méthode de Gauss-Seidel et fournir son implémentation en détail.

[ ]:
```
def gausseidel(A, b, x0, N, eps):
    """
    Résout un système d'équations linéaires Ax = b en utilisant la méthode␣
↪itérative de Gauss-Seidel.

    Args:
        A (numpy.ndarray): Matrice des coefficients du système linéaire.
        b (numpy.ndarray): Vecteur des termes constants du système linéaire.
        x0 (numpy.ndarray): Vecteur initial de l'approximation de la solution.
        N (int): Nombre maximum d'itérations.
        eps (float): Tolérance pour le critère d'arrêt basé sur la norme.

    Returns:
        tuple: Un tuple contenant :
            - x (numpy.ndarray): La solution approximative du système.
```

```
            - iterations (int): Le nombre d'itérations effectuées.

    Si la méthode ne converge pas, la fonction renvoie (0, 0).
    """
    n = len(A)  # Nombre de lignes/colonnes de la matrice A
    nrm = 0  # Initialisation de la norme
    for k in range(N):  # Itération jusqu'à atteindre le nombre maximum d'itérations (N)
        x = x0.copy()  # Copie du vecteur x0

        # Boucle sur chaque élément du vecteur x
        for i in range(n):
            S1 = 0  # Initialisation de la somme S1
            S2 = 0  # Initialisation de la somme S2

            # Boucle pour calculer la somme S1 (pour j < i)
            for j in range(i):
                S1 += A[i, j] * x[j]

            # Boucle pour calculer la somme S2 (pour j > i)
            for j in range(i + 1, n):
                S2 += A[i, j] * x0[j]

            # Calcul de la nouvelle valeur de x[i] en utilisant la formule de Gauss-Seidel
            x[i] = (b[i] - S1 - S2) / A[i, i]

        # Test d'arrêt : calcul de la norme du vecteur différence entre x et x0
        nrm = ((norme(x - x0) ** 2) / norme(x) ** 2)

        if nrm < eps:
            return x, k + 1  # Retourne la solution x et le nombre d'itérations nécessaires

        else:
            x0 = x  # Met à jour x0 avec la nouvelle valeur de x pour la prochaine itération

    print("La méthode de Gauss-Seidel ne converge pas")
    return 0, 0  # Retourne 0, 0 en cas de non-convergence
```

5.1 Test des deux fonctions

Après avoir mis en œuvre les méthodes de Jacobi et de Gauss-Seidel, il est temps de les soumettre à des tests rigoureux pour évaluer leurs performances et leur efficacité. Cette section présente les résultats de nos tests et compare les deux méthodes dans divers scénarios.

Nous examinerons comment ces méthodes résolvent des systèmes d'équations linéaires spécifiques et évaluerons leur rapidité de convergence, leur précision et leur stabilité. Les résultats de ces tests nous aideront à déterminer dans quelles situations chaque méthode est la plus appropriée.

Préparons-nous à explorer les performances des méthodes de Jacobi et de Gauss-Seidel dans les prochaines sections.

[ ]:
```python
import numpy as np

# Matrice 1
A1 = np.array([[1, -2, 2], [-1, 1, -1], [-2, -2, 1]])
x0 = np.array([1, 2, 3])
b = np.array([1, 2, 3])
N = 10
eps = np.finfo(float).eps

print("Test de la matrice 1:")
print("\nJacobi:")
x, k = jacobi(A1, b, x0, N, eps)
if isinstance(x, np.ndarray):
    print(f"Solution : {x}\nNombre d'itérations : {k}")


print("\nGauss-Seidel:")
x, k = gausseidel(A1, b, x0, N, eps)
if isinstance(x, np.ndarray):
    print(f"Solution : {x}\nNombre d'itérations : {k}")


# Matrice 2
A2 = np.array([[1, -1, -2], [-2, 1, 3], [0, 2, 1]])

print("\nTest de la matrice 2:")
print("\nJacobi:")
x, k = jacobi(A2, b, x0, N, eps)
if isinstance(x, np.ndarray):
    print(f"Solution : {x}\nNombre d'itérations : {k}")


print("\nGauss-Seidel:")
x, k = gausseidel(A2, b, x0, N, eps)
if isinstance(x, np.ndarray):
    print(f"Solution : {x}\nNombre d'itérations : {k}")
```

```
Test de la matrice 1:

Jacobi:
Solution : [-5. 10. 13.]
```

```
Nombre d'itérations : 3

Gauss-Seidel:
La méthode de Gauss-Seidel ne converge pas

Test de la matrice 2:

Jacobi:
La méthode de Jacobi ne converge pas

Gauss-Seidel:
Solution : [-14   7 -11]
Nombre d'itérations : 4
```

- Pour la première matrice, on remarque que la methode de Jacobi converge mais Gauss-Seidel ne converge pas
- Pour la deuxieme matrice on obtient le contraire des resultats precedants

6 Matrice Dominante Positive : Confirmation de la Convergence

La matrice que nous allons aborder est connue pour être dominante positive. Ce trait caractéristique suggère que la méthode itérative de Jacobi ou Gauss-Seidel devrait converger de manière fiable pour résoudre le système linéaire associé. Cependant, il est important de réaliser des tests pratiques pour confirmer cette convergence dans un contexte spécifique.

Dans cette section, nous allons mettre en œuvre la matrice et soumettre ces deux méthodes à un ensemble de tests rigoureux. L'objectif est de confirmer que la convergence s'applique réellement à ce cas particulier et de déterminer la rapidité de convergence ainsi que la précision des solutions obtenues.

6.1 Construction du matrice A :

[ ]:
```python
def matrice131(n) :
  A = np.zeros([n , n])
  for i in range(n) :
    A[i, i] = 3
    for j in range(n) :
      if j == i+1 or j == i - 1 :
        A[i, j] = A[j, i] = 1

  return A

A = matrice131(10)
print("A = \n", A)
```

```
A =
 [[3. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 3. 1. 0. 0. 0. 0. 0. 0. 0.]
```

```
[0. 1. 3. 1. 0. 0. 0. 0. 0. 0.]
[0. 0. 1. 3. 1. 0. 0. 0. 0. 0.]
[0. 0. 0. 1. 3. 1. 0. 0. 0. 0.]
[0. 0. 0. 0. 1. 3. 1. 0. 0. 0.]
[0. 0. 0. 0. 0. 1. 3. 1. 0. 0.]
[0. 0. 0. 0. 0. 0. 1. 3. 1. 0.]
[0. 0. 0. 0. 0. 0. 0. 1. 3. 1.]
[0. 0. 0. 0. 0. 0. 0. 0. 1. 3.]]
```

6.2 Resolution de l'equation par Jacobi

[ ]:
```python
A = matrice131(10)
b = np.ones(10)
x0 = np.ones(10)
# eps = np.finfo.eps
N = 100

x, k = jacobi(A, b, x0, N, eps)
if isinstance(x, np.ndarray):
    print(f"Solution : {x}\nNombre d'itérations : {k}")
```

```
Solution : [0.27638191 0.17085427 0.21105528 0.1959799  0.20100503 0.20100503
 0.1959799  0.21105528 0.17085427 0.27638191]
Nombre d'itérations : 46
```

6.3 Resolution de l'equation par Gausseidel

[ ]:
```python
A = matrice131(10)
b = np.ones(10)
x0 = np.ones(10)
# eps = np.finfo.eps
N = 100

x, k = gausseidel(A, b, x0, N, eps)
if isinstance(x, np.ndarray):
    print(f"Solution : {x}\nNombre d'itérations : {k}")
```

```
Solution : [0.27638191 0.17085427 0.21105527 0.1959799  0.20100502 0.20100503
 0.1959799  0.21105528 0.17085427 0.27638191]
Nombre d'itérations : 22
```

On observe que les deux methodes converge vers la meme solution mais la methode de *Gauss-Siedel* converge plus rapidement que celle de *Jacobi*

7 Relaxation

Nous avons examiné les méthodes itératives classiques de Jacobi et de Gauss-Seidel pour résoudre des systèmes d'équations linéaires, en mettant l'accent sur leur efficacité dans des contextes spéci-

fiques. Cependant, il existe une autre approche qui peut améliorer la convergence de ces méthodes dans certaines situations.

La méthode de relaxation, également connue sous le nom de méthode de sur-relaxation, est une technique itérative avancée qui vise à accélérer la convergence des méthodes classiques. Elle permet d'ajuster les valeurs mises à jour à chaque itération de manière à favoriser une convergence plus rapide et précise.

Dans cette section, nous allons explorer la méthode de relaxation, expliquer ses principes fondamentaux et son utilisation dans le contexte de la résolution de systèmes linéaires. Nous allons également comparer ses performances avec les méthodes de Jacobi et de Gauss-Seidel pour évaluer son impact sur la convergence.

Préparons-nous à plonger dans les détails de la méthode de relaxation et à découvrir comment elle peut être un atout précieux pour la résolution de systèmes linéaires.

7.1 Implementation

[ ]:
```python
def relaxation(A, b, x0, N, w, eps):
    """
    Résout un système linéaire Ax = b en utilisant la méthode de relaxation␣
 ↪avec Gauss-Siedel.

    Args:
        A (numpy.ndarray): Matrice des coefficients du système.
        b (numpy.ndarray): Vecteur des termes constants du système.
        x0 (numpy.ndarray): Vecteur initial d'estimations.
        N (int): Nombre maximal d'itérations.
        w (float): Facteur de relaxation (entre 0 et 2). Une valeur typique est␣
 ↪1 pour la méthode de Gauss-Seidel.
        eps (float): Tolérance pour la convergence.

    Returns:
        Tuple[numpy.ndarray, int]: Le vecteur de solution x et le nombre␣
 ↪d'itérations nécessaires pour atteindre la convergence.

    """
    n = len(A)  # Nombre de lignes de la matrice A
    for k in range(N):
        x = x0.copy()  # Copie du vecteur d'estimations initial
        xbar = x0.copy()  # Copie temporaire du vecteur d'estimations
        for i in range(n):
            S1 = 0  # Somme des termes à gauche de la diagonale
            S2 = 0  # Somme des termes à droite de la diagonale
            for j in range(i):
                S1 += A[i, j] * xbar[j]
            for j in range(i + 1, n):
                S2 += A[i, j] * x0[j]
            xbar[i] = ((b[i] - S1 - S2) / A[i, i])
```

```
            # Calcul de la nouvelle estimation x en utilisant le facteur de␣
↪relaxation w
            x[i] = (w * xbar[i] + (1 - w) * x0[i])

        # Calcul de la norme pour évaluer la convergence
        nrm = ((norme(x - x0) ** 2) / norme(x) ** 2)
        if nrm <= eps:
            # Convergence atteinte, retourne la solution et le nombre␣
↪d'itérations
            return x, k + 1
        else:
            x0 = x  # Mise à jour du vecteur d'estimations initial

    # Si la méthode ne converge pas après N itérations
    print("La méthode de relaxation ne converge pas")
    return 0, 0
```

7.2 Test du methode de relaxation

[ ]:
```
x0 = np.array([1, 2, 3], dtype=float)
b = np.array([1, 2, 3], dtype=float)
N = 500
eps = np.finfo(float).eps
w = 0.9
# print("Test du matrice 1 : ")


print("\n Matrice 1 : \n")
A = np.array([[1, -2, 2], [-1, 1, -1], [-2, -2, 1]], dtype=float)
x, k = relaxation(A, b, x0, 100, w, eps)
if isinstance(x, np.ndarray):
    print(f"Solution : {x}\nNombre d'itérations : {k}")

print("\n Matrice 2 : \n")
print("Si on prend omega = 1.1")
A2 = np.array([[1, -1, -2], [-2, 1, 3], [0, 2, 1]], dtype=float)
x, k = relaxation(A2, b, x0, N, w, eps)
if isinstance(x, np.ndarray):
    print(f"Solution : {x}\nNombre d'itérations : {k}")

print("\n Si on prend omega = 1.5 : ")
x, k = relaxation(A2, b, x0, N, 1.5, eps)
if isinstance(x, np.ndarray):
    print(f"Solution : {x}\nNombre d'itérations : {k}")

print("\n Si on prend omega = 0.5 : ")
```

```python
x, k = relaxation(A2, b, x0, N, 0.5, eps)
if isinstance(x, np.ndarray):
    print(f"Solution : {x}\nNombre d'itérations : {k}")
```

```
 Matrice 1 :

La méthode de relaxation ne converge pas

 Matrice 2 :

Si on prend omega = 1.1
Solution : [-14.00000001   7.         -11.        ]
Nombre d'itérations : 13

 Si on prend omega = 1.5 :
Solution : [-14.00000008   7.         -11.        ]
Nombre d'itérations : 40

 Si on prend omega = 0.5 :
Solution : [-14.00000018   7.         -11.00000001]
Nombre d'itérations : 35
```

- On remarque que la méthode ne converge pas pour la première matrice, car la méthode de Gauss-Seidel ne converge pas.
- De plus, on remarque que plus la valeur de (omega) s'approche de 1, moins d'itérations sont nécessaires pour atteindre la convergence.