Correction TP1

Module: Calcul Scientifique **Classes :** 3ème année **AU: 2023 / 2024**

TP1: Résolution numérique de systèmes d'équations linéaires

Applications numériques

Soient $n \geq 4$ et $a_i \in \mathbb{R}$ pour $i = 1, 2, 3$.
On s'intéresse à la résolution numérique d'un système linéaire de n équations écrit sous la forme (S_n): $AX = b$ avec

$$A = A(a_1, a_2, a_3) = \begin{pmatrix} a_1 & a_2 & 0 & \dots & \dots & 0 \\ a_3 & a_1 & a_2 & \ddots & & \vdots \\ 0 & a_3 & a_1 & a_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & \ddots & a_2 \\ 0 & \dots & \dots & 0 & a_3 & a_1 \end{pmatrix}, \quad X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} \quad \text{et} \quad b = \begin{pmatrix} 2 \\ 2 \\ \vdots \\ 2 \\ 2 \end{pmatrix}.$$


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

1. (a) Écrire une fonction `tridiag(a1,a2,a3,n)` qui renvoie la matrice tridiagonale $A = A(a_1, a_2, a_3)$ de taille n et le second membre b du système linéaire (S_n).

```
def tridiag(a1,a2,a3,n):
    A=a1*np.eye(n)+a2*np.diagflat(np.ones(n-1),1)+a3*...
    np.diagflat(np.ones(n-1),-1)
    b=2*np.ones((n,1))
    return A,b
```

(b) Tester la fonction `tridiag(a1,a2,a3,n)` pour $a_1 = 4$, $a_2 = a_3 = 1$ et $n = 10$.

```
a1,a2,a3,n=4,1,1,10
tridiag(a1,a2,a3,n)
```

array([[4., 1., 0., 0., 0., 0., 0., 0., 0., 0.], [1., 4., 1., 0., 0., 0., 0., 0., 0., 0.], [0., 1., 4., 1., 0., 0., 0., 0., 0., 0.], [0., 0., 1., 4., 1., 0., 0., 0., 0., 0.], [0., 0., 0., 1., 4., 1., 0., 0., 0., 0.], [0., 0., 0., 0., 1., 4., 1., 0., 0., 0.], [0., 0., 0., 0., 0., 1., 4., 1., 0., 0.], [0., 0., 0., 0., 0., 0., 1., 4., 1., 0.], [0., 0., 0., 0., 0., 0., 0., 1., 4., 1.], [0., 0., 0., 0., 0., 0., 0., 0., 1., 4.]])

2. (a) Écrire une fonction `matrice_diag_dominante(B)` prenant en entrée une matrice carrée B d'ordre n, qui vérifie si cette matrice est à diagonale strictement dominante ou non.

```
def matrice_diag_dominante(B):
    n=B.shape[0]
    etat=True
    i=0
    while ((i<n) and (etat==True)):
        etat=np.abs(B[i,i])>np.sum(np.abs(B[i,:]))-np.abs(B[i,i])
        i+=1
    return etat
```

(b) Tester la fonction `matrice_diag_dominante(B)` sur les matrices $B_1 = A(4,1,1)$ et $B_2 = A(1,1,1)$ pour $n = 10$.

```
B1=tridiag(4,1,1,10)[0]
B2=tridiag(1,1,1,10)[0]
print(matrice_diag_dominante(B1),matrice_diag_dominante(B2))
```

True False

3. (a) Écrire une fonction `jacobi(B, b1, X0, epsilon)` prenant en entrée une matrice carrée B d'ordre n, un second membre $b1$, une condition initiale `X0` et une tolérence `epsilon`, qui renvoie une solution approchée du SEL $BX = b1$ par la méthode de Jacobi et le nombre d'itérations effectuées.
La tolérance `epsilon` est utilisée pour le critère d'arrêt: $||BX^{(k)} - b1|| \leq$ `epsilon`.
On testera, à l'aide de la fonction `matrice_diag_dominante(B)`, si B est à diagonale strictement dominante, dans le cas contraire, on renverra B n'est pas à diagonale strictement dominante.

```
def jacobi(B, b1, X0, epsilon):
    etat=matrice_diag_dominante(B)
    if  etat==False:
```

```
        return ("B nést pas à diagonale strictement dominante")
    else:
        M=np.diagflat(np.diag(B))
        inv_M=np.linalg.inv(M)
        N=M-B
        D=inv_M.dot(N)
        C=inv_M.dot(b1)
        Niter_J=0
        while np.linalg.norm(B.dot(X0)-b1,1)>epsilon:
            X1=D.dot(X0)+C
            X0=X1
            Niter_J+=1
        return X1,Niter_J
```

(b) Utiliser la fonction `tridiag(a1,a2,a3,n)` pour tester la fonction de Jacobi avec les paramètres suivants

$n = 10,\ B = B_1,\ b1 = b,\ X^{(0)} = (1,1,\ldots,1)^t \in \mathcal{M}_{10,1}(\mathbb{R})$ et `epsilon` $= 10^{-6}$.

```
n=10
X0=np.ones((n,1))
epsilon=10**-6
b1=tridiag(a1,a2,a3,n)[1]
B=tridiag(a1,a2,a3,n)[0]
jacobi(B, b1, X0, epsilon)
```

(array([[0.42264915], [0.30940345], [0.33973712], [0.33164815], [0.33367039], [0.33367039], [0.33164815], [0.33973712], [0.30940345], [0.42264915]]), 24)

(c) En utilisant la fonction $np.linalg.solve(B_1, b)$, résoudre (S_{10}) et trouver l'erreur commise par la méthode de Jacobi en norme euclidienne.

```
Sol_exact=np.linalg.solve(B,b1)
Sol_exact
E_J=np.linalg.norm(Sol_exact-jacobi(B, b1, X0, epsilon)[0])
E_J
```

4.3384543293202557e-08

4. (a) Écrire une fonction `gauss_seidel(B, b1, X0, epsilon)` qui renvoie une solution approchée du SEL $BX = b1$ par la méthode de Gauss-Seidel et le nombre d'itérations effectuées en testant si B est à diagonale strictement dominante.

```
def gauss_seidel(B, b1, X0, epsilon):
    etat=matrice_diag_dominante(B)
    if  etat==0:
        return ('B nést pas à diagonale strictement dominante')
    else:
        M = np.tril(B)
        N = M-B

        invM=np.linalg.inv(M)
        A=invM.dot(N)
        C=invM.dot(b)
        Niter_GS=0

        while np.linalg.norm(B.dot(X0)-b1,1)>epsilon:
            X0=A.dot(X0)+C
            Niter_GS+=1
        return X0,Niter_GS
```

(b) Tester la fonction de Gauss-Seidel pour le même exemple considéré dans 3.b). Trouver, ensuite, l'erreur commise en norme euclidienne.

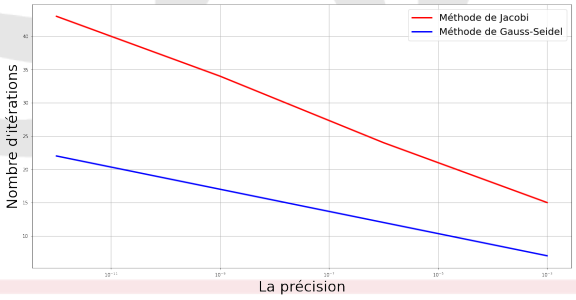
```
#test
gauss_seidel(B, b1, X0, epsilon)
#l'erreur
E_GS=np.linalg.norm(Sol_exact-gauss_seidel(B, b1, X0, epsilon)[0])
E_GS
```

(array([[0.42264909], [0.3094035], [0.33973705], [0.33164817], [0.33367035], [0.33367039], [0.33164812], [0.33973711], [0.30940344], [0.42264914]]), 12)
1.1203811979628228e-07

5. Comparer les résultats obtenus par les deux méthodes.
La méthode de Gauss-Seidel converge plus rapidement que celle de Jacobi, en effet, le nombre maximal d'itérations de Gauss-Seidel ($Niter_{GS} = 12$) est plus petit que le nombre maximal d'itérations de Jacobi ($Niter_J = 24$).
Par contre la méthode de Jacobi approche mieux la solution du système linéaire (S_{10}) que celle de Gauss-Seidel ($E_J \simeq 4.34 \times 10^{-8}$ et $E_{GS} \simeq 1.12 \times 10^{-7}$).

6. Représenter sur un même graphe, le nombre d'itérations par les deux méthodes itératives : Jacobi et Gauss-Seidel, en fonction de la précision `epsilon`. On considère `epsilon` $\in \{10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}\}$ et $n = 10$. Interpréter les résultats obtenus.

```
#Nombre d'itérations
epsilon=[10**-3,10**-6,10**-9,10**-12]
Iter_J=[]
Iter_GS=[]
for eps in epsilon:
    J=jacobi(B,b1,X0,eps)
    GS=gauss_seidel(B,b1,X0,eps)
    Iter_J.append(J[1])
    Iter_GS.append(GS[1])
#Représentation graphique
plt.figure(figsize=(20,10))
plt.plot(epsilon,Iter_J,'r',epsilon,Iter_GS,'b', linewidth=3,
markersize=12)
plt.xscale('log')
plt.xlabel('La précision',fontsize=30)
plt.ylabel('Nombre dítérations',fontsize=30)
plt.legend(('Méthode de Jacobi','Méthode de Gauss-Seidel'),
fontsize=20, loc = 0)
```



Pour $n = 10$, on observe que plus la précision epsilon est petite, plus le nombre maximal d'itérations de Gauss-Seidel et de Jacobi est grand, avec une convergence plus rapide de Gauss-Seidel que celle de Jacobi.

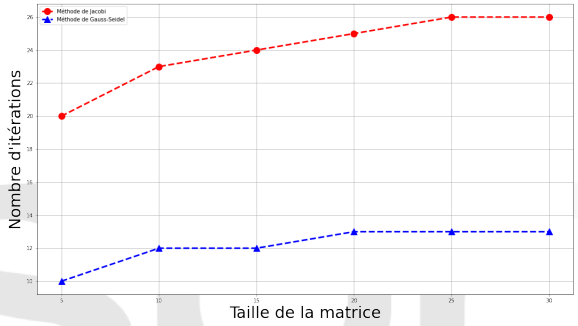
7. Soit la fonction `Niter(epsilon)` définie comme suit

```
def Niter(epsilon):
    N=np.arange(5,31,5)
    Niter_J=[]
    Niter_GS=[]
    for n in N:
        Ab=tridiag(4,1,1,n)
        X0=np.ones((n,1))
```

```
        J=jacobi(Ab[0],Ab[1],X0,epsilon)
        GS=gauss_seidel(Ab[0],Ab[1],X0,epsilon)
        Niter_J.append(J[1])
        Niter_GS.append(GS[1])
    return Niter_J, Niter_GS
```

Tester cette fonction pour `epsilon`=10^{-6}. Expliquer les résultats obtenus.

Pour expliquer les résultats obtenus, on pourra représenter sur un même graphe (voir la figure ci-dessous), le nombres d'itérations en fonction de la taille du système linéaire (S_n).



Pour epsilon=10^{-6}, on observe d'une part, que Jacobi nécessite plus d'itérations pour converger que Gauss-Seidel, d'autre part le nombre d'itérations des deux méthodes itératives croît légèrement en augmentant la taille n du système linéaire (S_n).