TP3correction

February 9, 2021

1 TP3 - Méthodes directes de résolution de Ax = b et méthodes itératives - Correction

[1]:
```
import numpy as np
from math import *
import matplotlib.pyplot as plt
from matplotlib import cm
```

2 Exercice 1 - Résolution de systèmes triangulaires

Définition de la méthode de descente

[2]:
```
def Lower(L,y):
    n = len(y)
    x = np.zeros(n)
    for i in range (n):
        x[i] = (y[i]-np.vdot(L[i,0:i],x[0:i])) / L[i,i]
    return x
```

On la teste sur un exemple :

[3]:
```
n=4
L=np.zeros((n,n))
for i in range (n):
    for j in range (i+1):
        L[i,j]=1+i+j
print('L=')
print(L)
y=np.random.rand(n)
print('y=',y)
print('Solution de Lx=y :')
x=Lower(L,y)
print(x)
print('erreur=',np.linalg.norm(L.dot(x)-y))
```

```
L=
[[1. 0. 0. 0.]
 [2. 3. 0. 0.]
```

```
 [3. 4. 5. 0.]
 [4. 5. 6. 7.]]
y= [0.51498932 0.12668419 0.62285509 0.59188483]
Solution de Lx=y :
[ 0.51498932 -0.30109815  0.05645595 -0.04304534]
erreur= 2.220446049250313e-16
```

2.0.1 Définition de la méthode de remontée

[4]:
```
def Upper(U,y):
    n = len(y)
    x = np.zeros(n)
    for i in range (n):
        x[n-1-i] = (y[n-1-i]-np.vdot(U[n-1-i,n-i:n],x[n-i:n])) / U[n-1-i,n-1-i]
    return x
```

3 Exercice 2 - Algorithme de décomposition LU

3.0.1 Définition de l'algorithme de décomposition LU

[5]:
```
def LU(A):
    n = len(A)
    for k in range (n-1):
        A[k+1:n,k] = A[k+1:n,k] / A[k,k] #on divise par le pivot sur la colonne␣
↪k à partir de l'indice i
        A[k+1:n,k+1:n] = A[k+1:n,k+1:n] - (np.transpose([A[k+1:n,k]])).
↪dot([A[k,k+1:n]]) #on traite la matrice extraite entre les indices k+1 et n-1
    return A
```

Attention ! Quand on définit un np.array avec des valeurs entières, python ne fait des opérations que dans les entiers. Ainsi pour la matrice $A = \begin{pmatrix} 14 & 4 \\ 5 & 7 \end{pmatrix}$, l'algorithme renverra la matrice $\begin{pmatrix} 14 & 4 \\ 0 & 7 \end{pmatrix}$, alors que la vraie décomposition compacte LU est $\begin{pmatrix} 14 & 4 \\ \frac{5}{14} & \frac{39}{7} \end{pmatrix}$. Pour palier ce problème d'opérations faites dans le monde des entiers, il faut, en définissant la matrice A : - soit mettre des flottants : `A = np.array([[14.,4.],[5.,7.]])` - soit dire qu'on manipule des flottants : `A = np.array([[14,4],[5,7]], dtype = 'float64')`

3.0.2 Définition de la matrice de Pascal

[6]:
```
def Pascal(n):
    P = np.ones((n,n))
    for i in range (1,n):
        for j in range (1,n):
            P[i,j] = P[i-1,j] + P[i,j-1]
    return P
```

Test pour n = 5

[7]:
```
n = 5
print('matrice de Pascal : ')
A=Pascal(n)
print(A)
LU(A)
print('LU :')
print(A)

U = np.zeros((n,n))
for i in range (0,n):
    U[i,i:n]=A[i,i:n]

L= A - U + np.diag(np.ones(n))

print('Décomposition LU de P :')
print('L=')
print(L)
print('U=')
print(U)
print('||P-LU||=',np.linalg.norm(Pascal(n)-L.dot(U)))
```

```
matrice de Pascal :
[[ 1.  1.  1.  1.  1.]
 [ 1.  2.  3.  4.  5.]
 [ 1.  3.  6. 10. 15.]
 [ 1.  4. 10. 20. 35.]
 [ 1.  5. 15. 35. 70.]]
LU :
[[1. 1. 1. 1. 1.]
 [1. 1. 2. 3. 4.]
 [1. 2. 1. 3. 6.]
 [1. 3. 3. 1. 4.]
 [1. 4. 6. 4. 1.]]
Décomposition LU de P :
L=
[[1. 0. 0. 0. 0.]
 [1. 1. 0. 0. 0.]
 [1. 2. 1. 0. 0.]
 [1. 3. 3. 1. 0.]
 [1. 4. 6. 4. 1.]]
U=
[[1. 1. 1. 1. 1.]
 [0. 1. 2. 3. 4.]
 [0. 0. 1. 3. 6.]
 [0. 0. 0. 1. 4.]
 [0. 0. 0. 0. 1.]]
```

```
||P-LU||= 0.0
```

4 Exercice 3 - Comparaison des méthodes de Jacobi, Gauss-Seidel et de relaxation

4.1 Question a)

On commence par la construction de la matrice M pour la méthode de Jacobi

[8]:
```
def MJ(A):
    M = np.diag(np.diag(A))    #on extrait de A sa diagonale et on forme une␣
↪matrice diagonale avec
    return M
```

Pour la méthode de Jacobi, les paramètres sont : la matrice A, le vecteur b et la condition initiale $x0$.

Il y a deux critères d'arrêt : la précision eps voulue sur $\|Ax0 - b\|$, et le nombre d'itérations maximum itmax.

[9]:
```
def Jacobi(A,b,x0,eps,itmax):
    x = x0    #on initialise avec x0
    M = MJ(A)
    N = M - A     #puisque N = E+F, -E et -F étant les parties sous- et␣
↪sur-diagonales de A
    err = np.linalg.norm(A.dot(x) - b) # l'erreur initiale est ||Ax0 - b||
    it = 0
    while (err > eps and it < itmax):  #on continue d'itérer tant qu'on n'a pas␣
↪atteint la précision eps souhaitée ou dépassé le nombre d'itérations
        x = Lower(M,N.dot(x) + b)  #on actualise x en trouvant la solution y de␣
↪My = Nx + b avec l'Exercice 1
        err = np.linalg.norm(A.dot(x) - b)    #on actualise l'erreur
        it = it + 1   #on actualise le paramètre d'itération
    return [x,it]   #on renvoie la solution approchée et le nombre d'itérations␣
↪nécessaires
```

On fait pareil pour la méthode de Gauss-Seidel :

[10]:
```
def MGS(A):
    M = np.diag(np.diag(A))
    for i in range (1,len(A)):   #on calcule M comme la somme des matrices␣
↪sous-diagonales
        M = M + np.diag(np.diag(A,-i),-i)
    return M
```

[11]:
```
def GaussSeidel(A,b,x0,eps,itmax):
    x = x0
    M = MGS(A)
```

```
    N = M - A
    err = np.linalg.norm(A.dot(x)-b)
    it = 0
    while err > eps and it < itmax:
        x = Lower(M,N.dot(x)+b) #on utilise encore la méthode de descente car M
↪est triangulaire inférieure
        err = np.linalg.norm(A.dot(x)-b)
        it = it + 1
    return [x,it]
```

On fait pareil pour la méthode de relaxation. Il y un paramètre supplémentaire.

[12]:
```
def MR(A,om):
    M = np.diag(np.diag(A)) / om
    for i in range (1,len(A)):
        M = M + np.diag(np.diag(A,-i),-i)
    return M
```

[13]:
```
def Relaxation(A,b,x0,om,eps,itmax):  #on a un paramètre omega supplémentaire
    x = x0
    M = MR(A,om)
    N = M - A
    err = np.linalg.norm(A.dot(x) - b)
    it = 0
    while err > eps and it < itmax:
        x = Lower(M,N.dot(x) + b)
        err = np.linalg.norm(A.dot(x) - b)
        it = it + 1
    return [x,it]
```

4.2 Question b)

On commence par construire la matrice A et le vecteur B.

[14]:
```
def Alaplace(n):
    A = 2 * np.diag(np.ones(n)) - np.diag(np.ones(n-1),-1) - np.diag(np.
↪ones(n-1),1)
    return A

def Btest(n):
    B = np.zeros(n)
    B[0] = 1
    B[n-1] = 1
    return B
```

[15]:
```
#Paramètres
n = 10
```

```
itmax = 100
om = 1.5
eps = 0 #ici on utilise seulement le second critère d'arrêt
x0 = np.zeros(n)
A = Alaplace(n)
B = Btest(n)
```

[16]:
```
#calcul des solutions approchées
[xJ,itJ] = Jacobi(A,B,x0,eps,itmax)
print('erreur Jacobi = ',np.linalg.norm(A.dot(xJ) - B), ", nombre d'itérations␣
↪=",itJ)
[xGS,itGS] = GaussSeidel(A,B,x0,eps,itmax)
print('erreur Gauss-Seidel = ',np.linalg.norm(A.dot(xGS) - B), ", nombre␣
↪d'itérations =",itGS)
[xR,itR] = Relaxation(A,B,x0,om,eps,itmax)
print('erreur Relaxation = ',np.linalg.norm(A.dot(xR) - B),", nombre␣
↪d'itérations =",itR)
```

```
erreur Jacobi =  0.0038447274751198255 , nombre d'itérations = 100
erreur Gauss-Seidel =  6.773832578676818e-05 , nombre d'itérations = 100
erreur Relaxation =  1.3287936532864002e-14 , nombre d'itérations = 100
```

4.3 Question c)

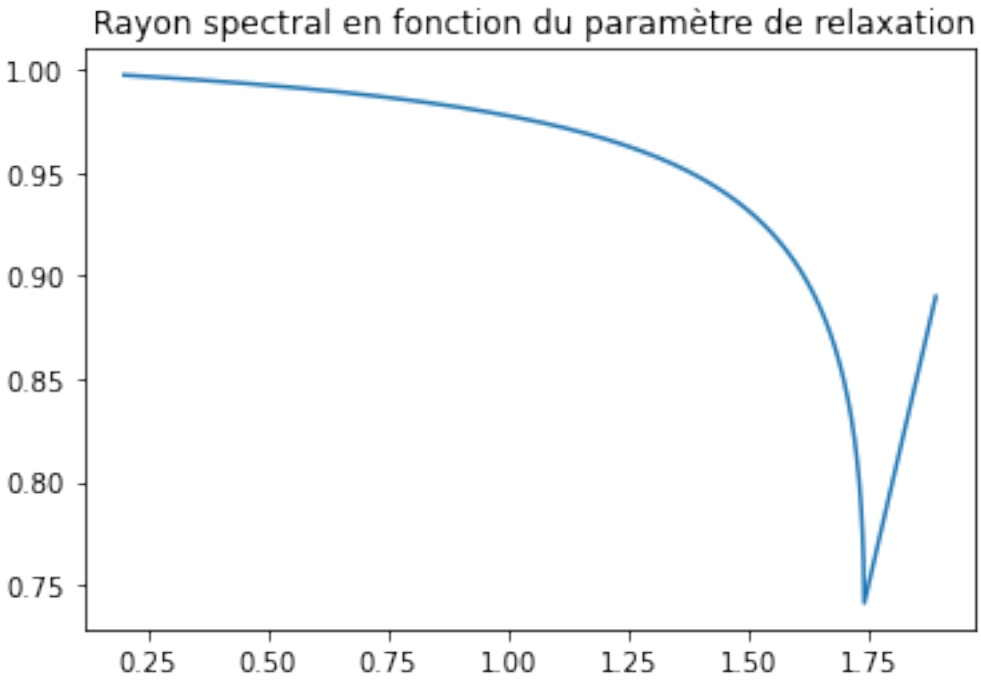
On commence par définir le rayon spectral :

[17]:
```
def rho(A):
    return np.linalg.norm(np.linalg.eigvals(A), inf) # norme infinie du vecteur␣
↪contenant toutes les valeurs propres
```

[18]:
```
#Paramètres
n = 20
itmax = 100
eps = 0
x0 = np.zeros(n)
#construction de la matrice A et du vecteur B
A = Alaplace(n)
B = Btest(n)
```

[19]:
```
#Vecteur pour les valeurs de omega et rho
om = np.linspace(0.2,1.89,1000)
#Calcul du rayon spectral de la matrice d'itérations
R = np.zeros(len(om))
for k in range (len(om)):
    M = MR(A,om[k])
    N = M - A
    R[k] = rho(np.linalg.inv(M).dot(N))
```

```
#Représentation graphique
plt. plot (om, R)
plt.title (" Rayon spectral en fonction du paramètre de relaxation")
plt.show()
```



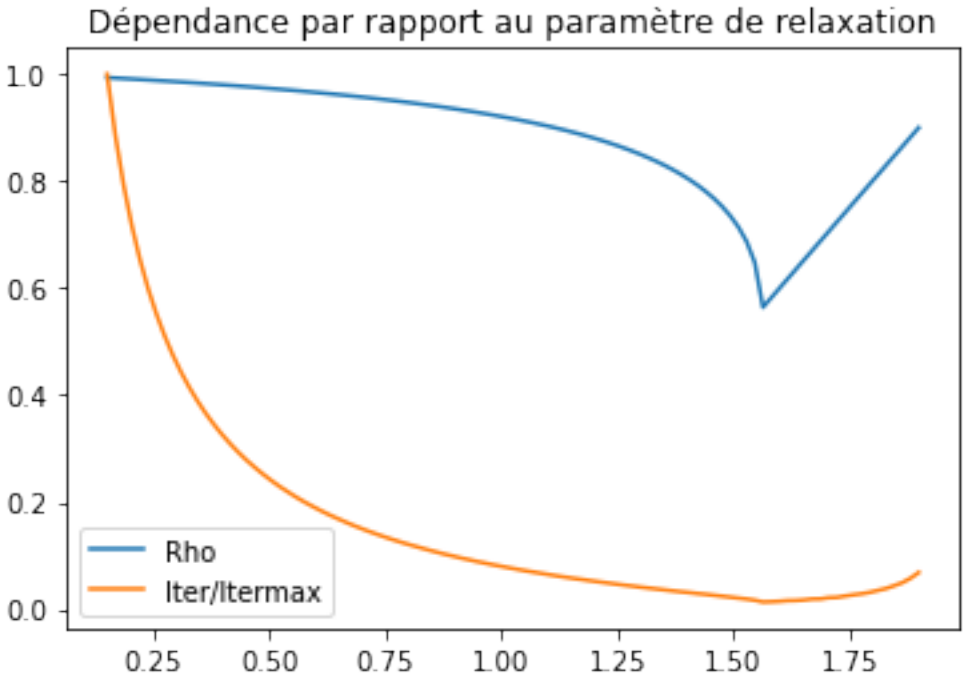
4.4 Question d)

[20]:
```
#Paramètres
itmax = 10**(4)
eps = 10**(-12)
n = 10
om = np.linspace(0.15,1.9,100)
Iter = np.zeros(len(om))
R = np.zeros(len(om))
A = Alaplace(n)
B = Btest(n)
x0 = np.zeros(n)
```

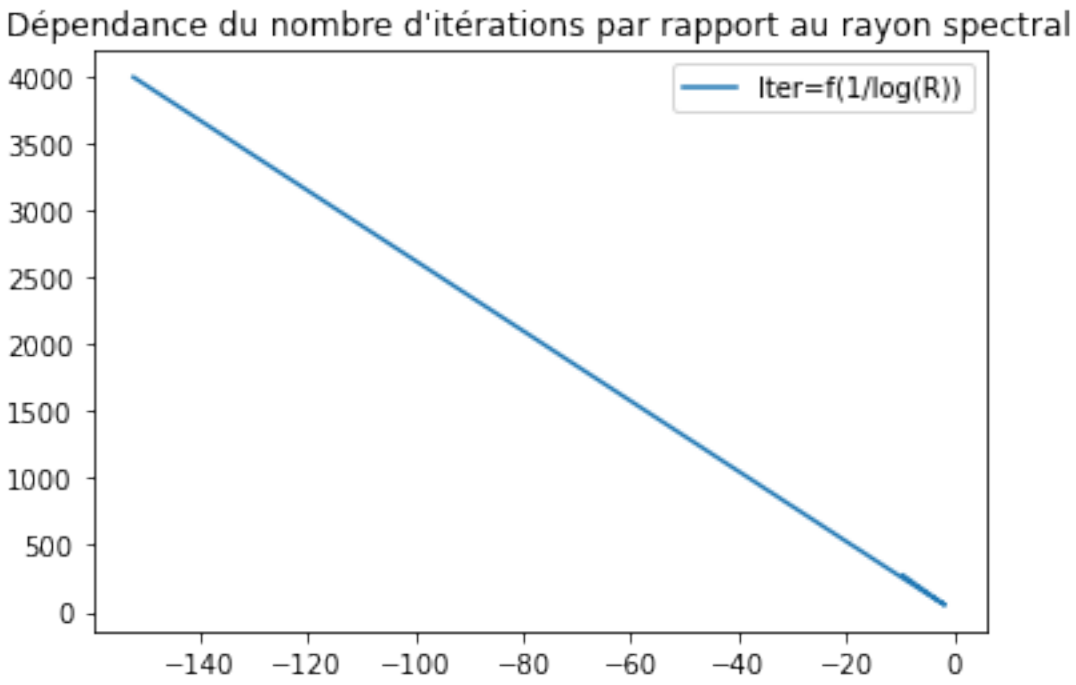
[21]:
```
#calcul des solutions avec la méthode de Jacobi
for k in range (len(om)):
    [xR,itR] = Relaxation(A,B,x0,om[k],eps,itmax)
    Iter[k] = itR
    M = MR(A,om[k])
    N = M - A
```

```
    R[k] = rho(np.linalg.inv(M).dot(N))
```

[22]:
```
#Évolution du rayon spectral et du nombre d'itérations
plt.plot(om,R,label = "Rho")
plt.plot(om,Iter/max(Iter),label = "Iter/Itermax")
plt.legend(loc = 'best')
plt.title("Dépendance par rapport au paramètre de relaxation")
plt.show()
```



[23]:
```
#Dépendance du nombre d'itérations par rapport au rayon spectral
plt.plot(np.log(R)**(-1), Iter,label = "Iter=f(1/log(R))")
plt.legend(loc = 'best')
plt.title("Dépendance du nombre d'itérations par rapport au rayon spectral")
plt.show()
```



On observe une dépendance linéaire. En effet, on a une estimation d'erreur du type $\epsilon_k = \|Ax_k - b\| \leq C\rho^k$, et donc la précision $\epsilon = 10^{-12}$ voulue est atteinte en $k \sim \frac{\log \epsilon}{\log \rho}$ itérations.

5 Exercice 4 - Résolution d'un problème de Poisson 2d

5.1 Question a)

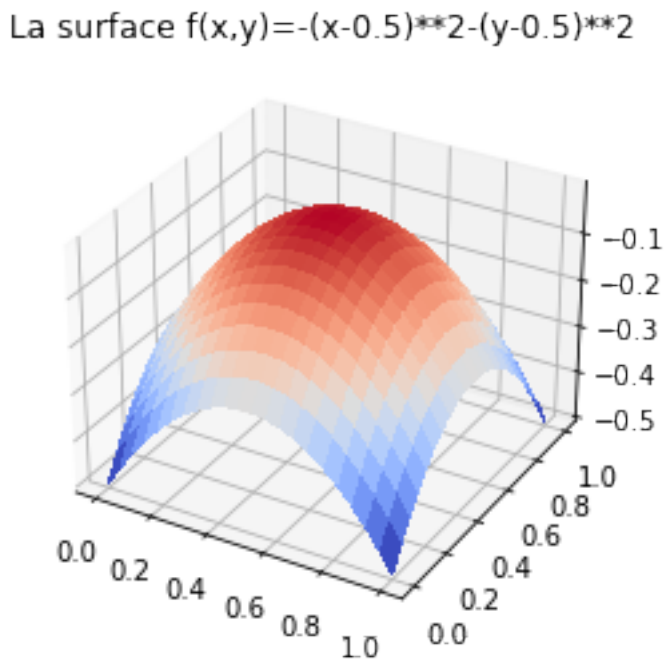
Définition de A : on représente $u_{i,j} = U[k]$ pour $k = 0, \ldots, N^2 - 1$ avec $k = N(i-1) + (j-1)$ pour $1 \leqslant i, j \leqslant N$. Il faut ensuite transposer l'équation aux différences centrées des indices (i, j) à l'indice k.

[24]:
```
def Alaplace2D(n):
    A = ((n+1)**2) * (-4*np.diag(np.ones(n**2)) + np.diag(np.ones(n**2-1),1)
                    + np.diag(np.ones(n**2-1),-1) + np.diag(np.ones(n**2-n),n) +␣
↪np.diag(np.ones(n**2-n),-n))
    return A
```

5.2 Question b)

[25]:
```
#Paramètres
n = 20
X = np.linspace(0,1,n+2)
Y = X
#Définition de la fonction f
def f(x,y):
```

```
    return -(x-0.5)**2 - (y-0.5)**2
#Surface enfendrée par f
fig = plt.figure()
ax = fig.gca(projection = '3d')
X, Y = np.meshgrid(X, Y)
Z = f(X,Y)
surf = ax.plot_surface(X, Y, Z,cmap = cm.coolwarm,linewidth = 0, antialiased =␣
 ↪False)
plt.title("La surface f(x,y)=-(x-0.5)**2-(y-0.5)**2")
plt.show()
```



[26]:
```
#Conversion matrice n*n et vecteur n**2
def mat2vec(A): #on convertit une matrice n*n en vecteur de taille n**2
    n = len(A)
    x = np.zeros(n**2)
    for i in range (n):
        x[n*i:n*(i+1)] = A[i,:]
    return x

def vec2mat(x): #on convertit un vecteur de taille n**2 en matrice nxn avec la␣
 ↪même convention que pour définir u
    N = len(x)
    n = floor(np.sqrt(N))
    A = np.zeros((n,n))
    for i in range (n):
```

```
        A[i,:] = x[n*i:n*(i+1)]
    return A
```

[27]:
```
#Résolution de l'équation de Poisson avec Jacobi
x0 = np.zeros(n**2) #donnée initiale
eps = 10**(-2) #précision souhaitée
itmax = 10**(5) #nombre max d'itérations
A = Alaplace2D(n) #on calcule la matrice A pour mettre le problème sous la␣
 ↪forme AU=F
F = mat2vec(Z[1:n+1,1:n+1]) #on calcule le second membre du système linéaire
[SolJ,itJ] = Jacobi(A,F,x0,eps,itmax) #ici on résoud le système avec la méthode␣
 ↪de Jacobi
ZsolJ = np.zeros((n+2,n+2))
#pour représenter la solution on convertit le vecteur fourni par la méthode de␣
 ↪Jacobi en matrice
ZsolJ[1:n+1,1:n+1] = vec2mat(SolJ)
#Représentation graphique
fig = plt.figure()
ax = fig.gca(projection = '3d')
surf = ax.plot_surface(X, Y, ZsolJ,cmap = cm.coolwarm,linewidth = 0,␣
 ↪antialiased = False)
plt.title("La solution de l'équation de Poisson avec Jacobi")
plt.show()
```