TP3

Exercice 1.
Soit un entier $n \geqslant 2$ et $h = 1/(n+1)$. On se donne un vecteur $f \in \mathbb{R}^n$ et on considère l'application J

$$u \in \mathbb{R}^n \longmapsto J(u) = \frac{1}{2h}\sum_{i=0}^{n}(u_{i+1}-u_i)^2 - h\sum_{i=1}^{n} u_i f_i,$$

où on a noté $u_0 = 0$ et $u_{n+1} = 0$.

1. Écrire $J(u)$ sous la forme

$$J(x) = \frac{1}{2h}\langle Ax, x\rangle - h\langle b, x\rangle,$$

où A est une matrice symétrique et b un vecteur à déterminer.

On a

$$J(x) = \frac{1}{2h}\langle Ax, x\rangle - h\langle b, x\rangle,$$

avec $A = tridiag(-1, 2, -1)$, $b = (f_1, \cdots, f_n)^T$.

Détails :

$$\sum_{i=0}^{n}(u_{i+1}-u_i)^2 = \sum_{i=0}^{n}[(u_{i+1}-u_i)u_{i+1} + (-u_{i+1}+u_i)u_i]$$

$$\sum_{i=0}^{n}(u_{i+1}-u_i)^2 = \sum_{i=0}^{n}(u_{i+1}-u_i)u_{i+1} + \sum_{i=0}^{n}(u_i-u_{i+1})u_i$$

$$\sum_{i=0}^{n}(u_{i+1}-u_i)^2 = \sum_{i=1}^{n+1}(u_i-u_{i-1})u_i + \sum_{i=0}^{n}(u_i-u_{i+1})u_i$$

On utilise le fait que $u_0 = u_{n+1} = 0$ et on a :

$$\sum_{i=0}^{n}(u_{i+1}-u_i)^2 = \sum_{i=1}^{n}(u_i-u_{i-1})u_i + \sum_{i=1}^{n}(u_i-u_{i+1})u_i$$

$$\sum_{i=0}^{n}(u_{i+1}-u_i)^2 = \sum_{i=1}^{n}(-u_{i-1}+2u_i-u_{i+1})u_i$$

Par conséquent

$$\nabla J(u) = \frac{1}{h}Ax - hb.$$

2. Montrer que le problème

$$\min_{u\in\mathbb{R}^n} J(u) \tag{1}$$

a une solution et une seule. On la notera $\bar{u}$.

3. Écrire "l'équation normale" associée à ce problème. Résoudre cette équation en utilisant la fonction `solve` du module `numpy.linalg`. On supposera que la solution obtenue est une très bonne approximation de la solution exacte $\bar{u}$.
Application : prendre pour f un vecteur dont toutes les composantes sont égales, $f_i = 1$ et $n = 10$. Afficher la courbe $((i\,h, u_i))_{i=0}^{n+1}$.
Équation normale :

$$A\bar{u} = h^2 b$$

```
import numpy as np
from numpy.linalg import solve
import matplotlib.pyplot as plt
n=10
h=1./n
D=np.zeros((n-1,n-1))
for i in range(n-2):
    D[i,i], D[i,i+1], D[i+1,i]= 2,-1,-1
D[n-2,n-2]=2
```

```
def f(x):
    return np.ones(x.shape)
```

```
X=np.linspace(0,1,n+1)
SolEquNor=solve(D,h*h*f(X[1:n]))
SolEquNor=list(SolEquNor)
SolEquNor.append(0)
SolEquNor.insert(0,0)
#Comparaison avec la solution exacte
SolEx=0.5*X*(1-X)
plt.plot(X,SolEquNor,label='Sol equ normales')
plt.plot(X,SolEx,label='Sol exacte')
plt.grid()
plt.legend()
plt.show()
```

4. On veut résoudre le problème de minimisation (1) par une méthode de gradient à pas constant dont les paramètres sont
   - τ : pas de la méthode,
   - η : la "tolérance" sur la gradient de la fonction à minimiser, au sens suivant : on arrête les itérations si la norme du gradient est plus petite que η,
   - `IterMax` : nombre maximal d'itérations autorisées : le programme s'arrête si on atteint ce nombre.

   On note $\tilde{u}$ la solution obtenue par la méthode du gradient. On fixe $n = 10$, $\eta = 10^{-4}$ et `IterMax=1000`. En faisant varier τ dans l'intervalle $[0.01, 0.07]$, tracer la fonction $\tau \mapsto$ nombre d'itérations effectuées par le programme. Ce nombre est donc égal à `IterMax` si la méthode ne converge pas. Indiquer la valeur de τ qui conduit au plus

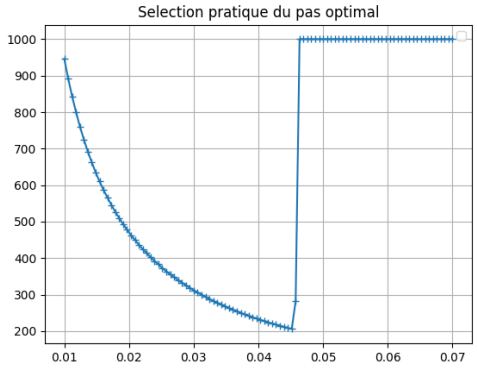
petit nombre d'itérations. On la notera $\tilde{\tau}$. Indiquer aussi la valeur minimale et la valeur maximale pour lesquelles l'algorithme n'a pas convergé. Comparer avec les résultats prévus par une étude théorique.

```
def GradientSimple(GradF,x0,rho,Tol,IterMax):
    x=x0
    gradfx=GradF(x)
    iter=0
    while  norm(gradfx) > Tol and iter <IterMax:
        x=x-rho*gradfx
        gradfx=GradF(x)
        iter+=1
    cvg = norm(gradfx) <= Tol
    return x,cvg,iter
```

```
def GradJ(x):
    y=np.dot(D,x)/h
    return y-h*f(x)
```

```
n,tau,Tol, IterMax=10, 0.005,1.e-2, 2000
x0=np.ones(n-1)
Sol,cvg,NbrIter=GradientSimple(GradJ,x0,tau,Tol,IterMax)
print('Pour tau=',tau, ' :: ',cvg,NbrIter)
Sol=list(Sol)
Sol.append(0)
Sol.insert(0,0)
plt.plot(X,Sol,label='Sol approchee')
plt.plot(X,SolEx,label='Sol exacte')
plt.grid()
plt.legend()
plt.show()
```

```
n,tau,Tol, IterMax=10, 0.5,1.e-4, 1000
x0=np.ones(n-1)
NbrIteration=[]
tauIntervalle =np.linspace(0.01, 0.07,100)
for tau in tauIntervalle:
    Sol,cvg,NbrIter=GradientSimple(GradJ,x0,tau,Tol,IterMax)
    NbrIteration.append(NbrIter)
plt.plot(tauIntervalle, NbrIteration,'+-')
plt.title("Selection pratique du pas optimal")
plt.grid()
plt.legend()
plt.show()
```



Par ailleurs, la théorie prédit la convergence pour $\tau < 2/L$. On calcule L par

```
L=norm(D,np.inf)/h
tauM=2/L
print(tauM)
0.05
```

5. Faire varier les différents paramètres : n, τ, η, `IterMax` et commenter les résultats observés, en particulier le comportement de l'erreur $\bar{u} - \tilde{u}$.
6. Reprendre l'étude avec, cette fois, $f_i = 3 - 6hi$. Commenter les résultats.

```
def f(z):
    return 3-6*X[1:len(X)-1]
```

```
n,tau,Tol, IterMax=40,0.01, 1.e-3, 5000
h=1./n
D=np.zeros((n-1,n-1))
for i in range(n-2):
    D[i,i], D[i,i+1], D[i+1,i]= 2,-1,-1
D[n-2,n-2]=2
X=np.linspace(0,1,n+1)
SolEx=X*(X-.5)*(X-1)
x=X[1:len(X)-1]
x0=np.ones(n-1)
Sol,cvg,NbrIter=GradientSimple(GradJ,x0,tau,Tol,IterMax)
print('Pour tau=',tau, ' :: ',cvg,NbrIter)
Sol=list(Sol)
Sol.append(0)
Sol.insert(0,0)
plt.plot(X,Sol,label='Sol approchee')
plt.plot(X,SolEx,label='Sol exacte')
plt.grid()
plt.legend()
plt.show()
```

Exercice 2.

On cherche à déterminer l'intersection des ellipses d'équations $x^2 + \frac{1}{2}y^2 = 1$ et $y^2 + \frac{1}{2}x^2 = 1$.

1. Tracer sur un même graphique les ellipses (on pourra les paramétrer)

```
from math import pi, sqrt
theta=np.linspace(0,2*pi)
x1,x2=np.cos(theta),sqrt(2)*np.sin(theta)
y1,y2=sqrt(2)*np.sin(theta),np.cos(theta)
plt.grid()
plt.plot(x1,y1,'b',x2,y2,'r')
plt.axis('equal')
plt.show()
```

2. Programmer la méthode de Newton pour résoudre le système d'équations. Choisir quatre initialisations différentes pour que l'algorithme converge vers chacun des quatre points d'intersection.

```
def Newton(F,JF,x0,Tol=1.e-8,IterMax=100):
    """
    methode de Newton pour resoudre F(x)=0
    F : R^n ---> R^n
    JF : jacobienne de F
    x0 dans R^n
    sortie : x, cvg, Niter
    """
    x=x0
    Fx=F(x)
    JFx=JF(x)
    z=solve(JFx,Fx)
    iter=0
    while  norm(z) > Tol and iter <IterMax:
        x=x-z
        Fx,JFx = F(x),JF(x)
        z=solve(JFx,Fx)
        iter+=1
    cvg = norm(z) <= Tol
    return x,cvg,iter
```

```
def F1(p):
    x, y = p[0], p[1]
    F1=x*x  +y*y/2-1
    F2=x*x/2+y*y  -1
    return np.array([F1,F2])
def JF1(p):
    x, y  = p[0], p[1]
    J=np.zeros((2,2))
    J[0,:]=[2*x,y]
    J[1,:]=[x,2*y]
    return J
#
x0=np.array([-1,1])
```

```
Tol,IterMax=1.e-4,1000
Sol, Cvg, Iterations =Newton(F1,JF1,x0,Tol,IterMax)
if Cvg:
    print('La methode de Newton converge')
    print('x = ',Sol)
    print('F(x) = ',norm(F1(Sol)))
    print('nombre d operations = ',Iterations)
else:
    print('La methode de Newton ne converge pas !!')
    print('x = ',Sol)
    print('F(x) = ',norm(F1(Sol)))
```

Exercice 3.

L'équation normale associée au problème de minimisation

$$\min_{\|x\|_2=1} \langle Ax, x\rangle$$

où $A \in \mathcal{M}_n(\mathbb{R})$ est une matrice symétrique peut s'écrire

$$Ax + \lambda x = 0.$$

On cherche à déterminer le vecteur x et le multiplicateur de Lagrange λ (qui est donc l'opposé d'une valeur propre de A) en résolvant, par la méthode de Newton, le système d'équations

$$Ax + \lambda x = 0, \quad \|x\|_2^2 = 1.$$

1. Résoudre ce problème pour la matrice $A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 2 & 1 \\ 3 & 1 & 3 \end{pmatrix}$.

```
def F2(p):
    x, y, z, l= p[0], p[1], p[2], p[3]
    F=np.zeros((4,1))
    w=np.dot(A,[x,y,z])-l*[x,y,z]
    F[0]=w[0]
    F[1]=w[1]
    F[2]=w[2]
    F[3]=x*x+y*y+z*z-1
    return F
def JF2(p):
    x, y, z, l= p[0], p[1], p[2], p[3]
    J=np.zeros((4,4))
    J[0,0], J[0,1], J[0,2], J[0,3]=A[0,0]-l,A[0,1]  ,A[0,2]  ,-x
    J[1,0], J[1,1], J[1,2], J[1,3]=A[1,0]  ,A[1,1]-l,A[1,2]  ,-y
    J[2,0], J[2,1], J[2,2], J[2,3]=A[2,0]  ,A[2,1]  ,A[2,2]-l,-y
    J[3,0], J[3,1], J[3,2], J[3,3]=2*x     , 2*y    , 2*z    , 0
    return J
```

```
A=np.array([[1,2,3],[2,2,1],[3,1,3]])
x0=np.ones((4,1))
Tol,IterMax=1.e-4,1000
Sol, Cvg, Iterations =Newton(F2,JF2,x0,Tol,IterMax)
vec,vap=Sol[0:2],Sol[2]
if Cvg:
    print('La methode de Newton converge')
    print('x = ',Sol)
    print('F(x) = ',norm(F2(Sol)))
    print('nombre d operations = ',Iterations)
else:
    print('La methode de Newton ne converge pas !!')
    print('x = ',Sol)
    print('F(x) = ',norm(F2(Sol)))
```

2. Vérifier le résultat obtenu on calculant le spectre de cette matrice.

```
from numpy.linalg import eig  # pour v.p
D, V = eig(A)
print(D)
```

Exercice 4.
On considère un chemin de classe $\mathscr{C}^1$

$$\gamma : t \in [0,1] \longmapsto \gamma(t) = (x(t), y(t)) \in \mathbb{R}^2$$

conduisant du point de départ $\gamma(0) = (0,0)$ au point d'arrivée $\gamma(1) = (1,1)$. Le but de cette étude est de minimiser

$$H(\gamma) = \frac{1}{2}\int_0^1 [x'(t)^2 + y'(t)^2]dt \tag{2}$$

sous la contrainte suivante : le chemin doit contourner un obstacle qu'on suppose être un disque entièrement compris dans le carré unité du plan

$$D = \{(x,y) \in \mathbb{R}^2, (x-a)^2 + (y-b)^2 \leqslant r^2\}.$$

Pour résoudre ce problème, nous allons "pénaliser" la contrainte. Pour $\varepsilon > 0$ et "petit", on cherche à minimiser

$$H_\varepsilon(\gamma) = H(\gamma) + \frac{1}{\varepsilon}R(\gamma)$$

avec

$$R(\gamma) = \frac{1}{2}\int_0^1 \max\left(0, r^2 - (x-a)^2 - (y-b)^2\right)^2 dt. \tag{3}$$

Noter que le terme de "pénalisation" n'intervient que si le point $(x(t), y(t))$ est situé dans l'obstacle. Si aucun point du chemin γ n'est situé dans l'obstacle, on a $H_\varepsilon(\gamma) = H(\gamma)$.

Discrétisation de (2). Pour $N \in \mathbb{N}^*$, on pose $h = 1/N$ et $t_n = nh$ pour $n = 0, \cdots, N$. Pour calculer l'intégrale $H(\gamma)$, on écrit

$$H(\gamma) = \frac{1}{2} \sum_{n=0}^{N-1} \int_{t_n}^{t_{n+1}} [x'(t)^2 + y'(t)^2] dt$$

et fait l'approximation

$$H(\gamma) \simeq \mathscr{H}(x, y) = \frac{1}{2} \sum_{n=0}^{N-1} \left[\left(\frac{x_{n+1} - x_n}{h} \right)^2 + \left(\frac{y_{n+1} - y_n}{h} \right)^2 \right] h,$$

où $x = (x_i)_{i=1}^{N-1} \in \mathbb{R}^{N-1}$, $y = (y_i)_{i=1}^{N-1} \in \mathbb{R}^{N-1}$ et les x_j (resp. y_j) sont des approximations des $x(t_j)$ (resp. $y(t_j)$ avec $(x_0, y_0) = \gamma(0)$, $(x_N, y_N) = \gamma(1)$.

Discrétisation de (3). On fait l'approximation

$$R(\gamma) \simeq \mathscr{R}(x, y)$$

avec

$$\mathscr{R}(x, y) = \frac{h}{2} \sum_{n=0}^{N-1} \left(\max(0, r^2 - (x_n - a)^2 - (y_n - b)^2) \right)^2.$$

On cherche donc à minimiser

$$\mathscr{H}_\varepsilon(x, y) = \mathscr{H}(x, y) + \frac{1}{\varepsilon} \mathscr{R}(x, y), \tag{4}$$

pour x et y dans $\mathbb{R}^{N-1}$.

1. Montrer que

   $$\mathscr{H}(x, y) = \frac{1}{2h} \left(\langle Ax, x \rangle + \langle Ay, y \rangle - 2\langle \bar{x}, x \rangle - 2\langle \bar{y}, y \rangle \right) + \text{une constante},$$

   où $A \in \mathcal{M}_{N-1}(\mathbb{R})$, $\bar{x} \in \mathbb{R}^{N-1}$ et $\bar{y} \in \mathbb{R}^{N-1}$ sont à déterminer.

   $A = \text{tridiag}(-1, 2, -1)$, $\bar{x} = (x_0, 0, \cdots, 0, x_N)^T$ et $\bar{y} = (y_0, 0, \cdots, 0, y_N)^T$. On a

   $$2h\mathscr{H}(x, y) = \langle Ax, x \rangle - 2\langle x, \bar{x} \rangle + \langle Ay, y \rangle - 2\langle y, \bar{y} \rangle + \text{const},$$

   le terme constant étant égal à $x_0^2 + y_0^2 + x_N^2 + y_N^2$.

2. Calculer les gradients de la fonction $\mathscr{H}(x, y)$ par rapport à x et par rapport à y

   $$\nabla_x \mathscr{H}(x, y) = \left(\frac{\partial \mathscr{H}}{\partial x_n}(x, y) \right)_n \in \mathbb{R}^{N-1}, \quad \nabla_x \mathscr{H}(x, y) = \left(\frac{\partial \mathscr{H}}{\partial y_n}(x, y) \right)_n \in \mathbb{R}^{N-1}$$

   $$\nabla_x \mathscr{H}(x, y) = \frac{1}{h} (Ax - \bar{x}), \quad \nabla_y \mathscr{H}(x, y) = \frac{1}{h} (Ay - \bar{y}).$$

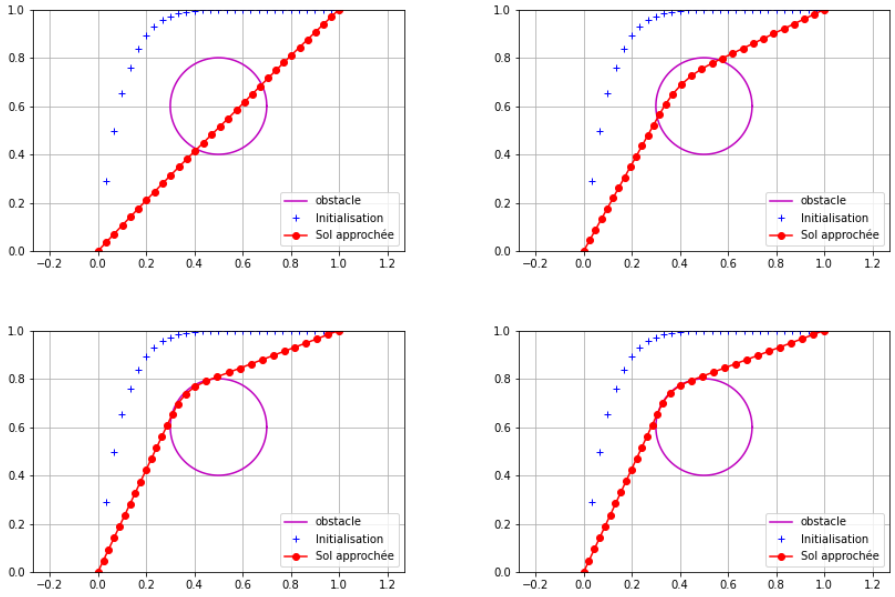


FIGURE 1 – Solutions du problème de minimisation (4) pour différentes valeurs du paramètre de pénalisation $\varepsilon \in \{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$. Si ε n'est pas trop petit, la solution minimale reste la ligne droite, même s'il faut payer un prix en passant par l'obstacle. Si ε est assez petit, le minimum ne peut plus passer par l'obstacle, ce serait trop pénalisant. Exemples de paramètres pour la simulation : $N = 30$, obstacle (disque centré en $(0.5, 0.6)$ et de rayon 0.2), méthode du gradient ($\eta = 10^{-4}$, $\tau = 1/10.$, `IterMax=?`), initialisation : $(x_i, y_i), y_i = 1 - (1 - x_i)^{10}$.

3. Calculer les gradients de la fonction $\mathscr{R}(x, y)$ par rapport à x et par rapport à y

   En dehors de l'obstacle, $\mathscr{L} \equiv 0$ et dans l'obstacle, on a

$$\begin{pmatrix} \frac{\partial \mathscr{L}}{\partial x_n} \\ \frac{\partial \mathscr{L}}{\partial y_n} \end{pmatrix}(x, y) = -2h(r^2 - (x_n - a)^2 - (y_n - b)^2) \begin{pmatrix} x_n - a \\ y_n - b \end{pmatrix}.$$

4. Utiliser la méthode du gradient à pas fixe pour résoudre le problème. Reproduire un graphique comme celui de la figure 1.

5. On fixe $N = 30, \eta = 10^{-4}, \tau = 1, \varepsilon = 1/10$, `IterMax=5000`. Calculer la solution obtenue par le programme. Diviser ε par 2 et partir de la solution précédente comme donnée initiale de l'algorithme du gradient. Répéter plusieurs fois ce processus. Commenter.

6. **Facultatif**
   Voici quelques idées de développements.
   (a) Modifier la forme de l'obstacle en considérant un carré ou un losange, ...
   (b) Rajouter un autre obstacle.
   (c) ...

```
def LaMatrice(n):
    D=np.zeros((n-1,n-1))
    for i in range(n-2):
        D[i,i], D[i,i+1], D[i+1,i]= 2,-1,-1
    D[n-2,n-2]=2
    return D
```

```
def J(p):
    x,y=p[0],p[1]
    z =np.dot(np.dot(D,x),x)-2*np.dot(x,Hor)
    z+=np.dot(np.dot(D,y),y)-2*np.dot(y,Ver)
    c=norm(Depart)**2+norm(Arrivee)**2
    return 0.5*(z+c)/h
```

```
def GradJ(p):
    z1=(np.dot(D,p[0])-Hor)/h
    z2=(np.dot(D,p[1])-Ver)/h
    return [z1,z2]
```

```
def GradJP1(p):
    dx,dy=GradJ(p)
    p0,p1=p[0],p[1]
    test=DistanceObstacle(p)
    for k in range(len(test)):
        if test[k]>=0:
            dx[k]=dx[k]-2*h/eps1*test[k]*(p0[k]-Centre1[0]);
            dy[k]=dy[k]-2*h/eps1*test[k]*(p1[k]-Centre1[1])
    return [dx,dy]
```

```
def JP1(p):
    z=J(p)
    test=DistanceObstacle(p)
    for k in range(len(test)):
        if test[k]>=0:
            z+=h/eps1*test[k]*test[k]
    return z
```

```
def DistanceObstacle(p):
    p0,p1=p[0],p[1]
    n=len(p0)
    test=np.zeros(n-1)
    for k in range(n-1):
        test[k]=Rayon1**2-(p0[k]-Centre1[0])**2-(p1[k]-Centre1[1])**2
    return test
```

```
def GradientRobot(GradF,x0,rho,Tol,IterMax):
    x=x0
    gradfx=GradF(x0)
```

```
    iter=0
    while  norm(np.concatenate(gradfx)) > Tol and iter <IterMax:
        x[0]=x[0]-rho*gradfx[0]
        x[1]=x[1]-rho*gradfx[1]
        gradfx=GradF(x)
        iter+=1
    cvg = norm(np.concatenate(gradfx)) <= Tol
    return x,cvg,iter
```

```
# Depart et arrivee
Depart,Arrivee=[0,0],[1,1]
# Discretisation
n=30;h=1/n
# Matrice du "faux laplacien"
D=LaMatrice(n)
# utile pour le grad
Hor=np.zeros(n-1);Hor[0],Hor[-1]=Depart[0],Arrivee[0]
Ver=np.zeros(n-1);Ver[0],Ver[-1]=Depart[1],Arrivee[1]
# Initialisation
x0=  np.linspace(0,1,n+1);x0[0]=Depart[0];x0[n]=Arrivee[0]
y0=1-(1-x0)**10;          y0[0]=Depart[1];y0[n]=Arrivee[1]

init=[0,0]
init[0]=x0[1:len(x0)-1]
init[1]=y0[1:len(y0)-1]
# Parametres methode du gradient
init0=init[:]
Init0=list(init0[0]);Init0.append(Arrivee[0]);Init0.insert(0,Depart[0])
Init1=list(init0[1]);Init1.append(Arrivee[1]);Init1.insert(0,Depart[1])
# Un seul run pour illustrer l'enonce
Tol, IterMax=1.e-4, 1000
L=norm(D,np.inf)/h
tauM=2/L
print(tauM)
tau=.01
eps=.00001
# un seul obstacle
Centre1=[0.5,0.6];Rayon1=0.2;eps1=eps
from math import pi as po
theta =np.linspace(0,2*po,100)
x=Centre1[0]+Rayon1*np.cos(theta);y=Centre1[1]+Rayon1*np.sin(theta)

Sol,cvg,Niter= GradientRobot(GradJP1,init[:],tau,Tol,IterMax)
Sol0=list(Sol[0]);Sol0.append(Arrivee[0]);Sol0.insert(0,Depart[0])
Sol1=list(Sol[1]);Sol1.append(Arrivee[1]);Sol1.insert(0,Depart[1])
plt.plot(x,y,'m',label='obstacle')
plt.plot(Init0,Init1,'+b' ,label='Initialisation')
plt.plot(Sol0,Sol1  ,'o-r',label='Sol approchee')
plt.grid()
plt.axis('equal');
```

```
plt.xlim(Depart[0],Arrivee[0])
plt.ylim(Depart[1],Arrivee[1])
plt.legend()
plt.show()
```

Exercice 5. (pour continuer) Retour sur le gradient à pas optimal.
On considère la fonction f définie pour tout $v = (v_1, v_2) \in \mathbb{R}^2$ par

$$f(v) = (v_1 - 4)^2 + 2(v_2 - 3)^2 + v_1 v_2.$$

1. Montrer que f admet un unique minimiseur sur $\mathbb{R}^2$ et le déterminer.
2. Implémenter la fonction f qui prend en argument un vecteur `numpy` de taille 2, représentant un point v de $\mathbb{R}^2$, et qui retourne la valeur de la fonction f en ce point.

```
def f(x):
    return (x[0]-4)**2 + 2*(x[1]-3)**2 + x[0]*x[1]
```

3. Tracer les lignes de niveaux de f.

```
xx = np.linspace(1,4,100)
yy = np.linspace(1,4,100)
xx,yy = np.meshgrid(xx,yy)
zz = np.zeros(xx.shape)
for i in range(xx.shape[0]):
    for j in range(xx.shape[1]):
        zz[i,j] = f(np.array([xx[i,j],yy[i,j]]))
plt.title("Lignes de niveau de $f$")
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.contourf(xx,yy,zz)
plt.colorbar()
plt.show()
```

4. Implémenter la fonction `gradf` qui calcule le gradient de f en un point.

```
def gradf(x):
    return np.array([ 2*(x[0]-4)+x[1], 4*(x[1]-3)+x[0]])

# TEST
x = np.ones(2)
print(gradf(x))
```

5. On rappelle que dans la méthode de gradient à pas optimal, le pas ρ_n change à chaque itération :

$$v_{n+1} = v_n - \rho_n \nabla f(v_n) \quad \text{où } \rho_n = \arg\min_{\rho > 0} f(v_n - \rho \nabla f(v_n))$$

Posons $g_n(\rho) = f(v_n - \rho\nabla f(v_n))$, définie de $\mathbb{R}$ dans $\mathbb{R}$. A chaque itération, pour déterminer le pas optinal ρ_n, nous cherchons la solution du problème (possiblement non linéaire) suivant :

$$g_n'(\rho) = 0$$

Montrer, pour tout $n \geqslant 0$, que la dérivée de g_n est

$$\forall \rho \in \mathbb{R} \quad g_n'(\rho) = -\nabla f(v_n) \cdot \nabla f(v_n - \rho\nabla f(v_n)).$$

6. Implémenter une fonction `gprime` qui prend en arguments un réel ρ, la fonction `gradf` et un point $v \in \mathbb{R}^2$ (vecteur `numpy` de taille 2) et qui retourne la valeur $g'(\rho)$ qui dépend de v et $\nabla f(v)$.

```
def gprime(rho, gradf, x):
    return -np.dot(gradf(x-rho*gradf(x)), gradf(x))


#TEST
print(gprime(0.1,gradf,x))
```

7. Afin de déterminer une valeur approchée de la solution $g_n'(\rho) = 0$, on va considérer une variante de la méthode de Newton : dans la formule de Newton donnée par

$$\rho_{k+1} = \rho_k - \frac{g_n'(\rho_k)}{g_n''(\rho_k)},$$

pour ne pas avoir à calculer g_n'', on remplace $g_n''(\rho_k)$ par le taux d'accroissement :

$$\frac{g_n'(\rho_k + \delta) - g_n'(\rho_k)}{\delta}, \quad \text{avec} \quad \delta = 10^{-8}.$$

Ainsi, à chaque étape n de l'algorithme du gradient, on construit la suite

$$\begin{cases} \rho_0 \in \mathbb{R} & \text{donné} \\ \rho_{k+1} & = \rho_k - \delta \dfrac{g_n'(\rho_k)}{g_n'(\rho_k + \delta) - g_n'(\rho_k)}. \end{cases}$$

qui converge vers ρ_n, la solution du problème $\rho_n = \arg\min_{\rho>0} f(v_n - \rho\nabla f(v_n))$.

Programmer une fonction `newton` qui prend en argument
— la fonction `gprime`,
— la fonction `gradf`,
— x, un vecteur `numpy` de taille 2,
— une valeur initiale $\rho_0 \in \mathbb{R}$,
— une tolérance ε,
— un nombre maximum d'itérations $N_{\max}$
et qui retourne un réel ρ^*, solution de l'équation $g'(\rho) = 0$, en utilisant la variante de l'algorithme de Newton donnée.
On testera cette fonction avec les valeurs suivantes :

$$x = 0_{\mathbb{R}^2}, \quad \rho_0 = 1, \quad \varepsilon = 1e-5, \quad N_{\max} = 100$$

Indication : On choisira soigneusement le critère d'arrêt pour cet algorithme de Newton et on ajoutera un compteur pour arrêter l'algorithme si le nombre d'itérations dépasse N_{max}. Ce compteur sera également retourné par la fonction.

```
def newton(gprime, gradf, x, rho0, eps=1e-5, Nmax=100):
    rho = rho0
    n = 0
    delta = 1e-8
    while np.abs(gprime(rho,gradf,x))>eps and n<Nmax:
      gp = gprime(rho,gradf,x)
      rho = rho - delta*gp/(gprime(rho+delta,gradf,x) - gp)
      n += 1
    return rho, n

# Test
x = np.zeros(2)
rho0 = 1
eps = 1e-5
Nmax = 100
rho, n = newton(gprime, gradf, x, rho0, eps, Nmax)
print(rho,n)
```

8. Programmer la fonction `gradopt` qui prend en arguments :
   — v_0, l'initialisation de l'algorithme,
   — ρ_0, un pas initial,
   — ε, la précision,
   — la fonction `gradf`,
   — la fonction `gprime`,
   — Nmax, le nombre d'itérations maximum
   — $\varepsilon_{\text{Newton}}$, la précision de l'algorithme de Newton,
   — $N_{\text{max, Newton}}$, le nombre maximum de l'algorithme de Newton,

   et qui retourne le minimiseur recherché et le nombre d'itérations de la méthode de gradient à pas optimal et le nombre total d'itérations de l'algorithme de Newton, en utilisant l'algorithme de Newton à chaque étape pour déterminer le pas optimal.

   Indication : On choisira soigneusement le critère d'arrêt pour cet algorithme de gradient.

```
def gradient_optimal(gradf, gprime, x0, rho0, eps=1e-6,
                eps_Newton=1e-6, Nmax=500, Nmax_Newton=100):

    x=x0.copy()
    xlist=[x]
    n=0
    alpha = rho0
    Nnew = 0
    while (norm(gradf(x))>eps and (n<Nmax)):
        alpha, n_newton=newton(gprime, gradf, x, alpha,
                               eps_Newton, Nmax_Newton)
```

```
        x=x-alpha*gradf(x)
        n += 1
        xlist.append(x)
        Nnew +=  n_newton
    return x, n, Nnew, np.array(xlist)
```

9. Tester la méthode du gradient à pas optimal sur la fonction f en traçant le nombre d'itérations en fonction du paramètre ρ_0. On comptera aussi le nombre d'itérations total (nombre d'itérations de la méthode de gradient à pas optimal plus le nombre total d'itération de l'algorithme de Newton). On prendra les valeurs suivantes pour les autres paramètres :

$$v_0 = (1,1), \quad \varepsilon = 1e-6, \quad N_{\max} = 500, \quad \varepsilon_{Newton} = 10^{-6}, \quad N_{\max,\text{Newton}} = 10.$$

Commenter le graphique obtenu (et pour $\varepsilon_{Newton} = 10^{-12}$?).
Indication : On fera varier ρ_0 entre 0.01 et 1 par pas de 0.01.

```
alpha = np.arange(1e-8,1e-6,1e-8)
N = np.zeros(alpha.size)
Nnew = np.zeros(alpha.size)
for i in range(N.size):
   init = np.array([1,1])
    x, N[i], Nnew[i], xlist=gradient_optimal(gradf,gprime,init,alpha[i])

plt.plot(alpha,N, 'b')
plt.plot(alpha,Nnew, 'r')
plt.title(r"Nombre d'iterations en fonction de
        $\rho0$ pour $\varepsilon_{Newton}=10^{-6}$")
plt.xlabel(r"Pas initial $\rho_0$")
plt.ylabel(r"Nombre d'iterations")
plt.show()
print("Nombre minimum d'iterations : {:}".format(N.min()))
```

10. Modifier la fonction `gradopt` afin qu'elle retourne également la liste des points construits lors de l'algorithme du gradient. Tracer ensuite les points de cette suite sur le graphique des lignes de niveau de f réalisé à la question 3. Comparer avec la méthode de gradient à pas constant.

```
def gradient_descent(gradf, x0, alpha, eps=1e-6, Nmax=500):
    x=x0
    n=0
    xlist=[x]
    while (norm(gradf(x))>eps and (n<Nmax)):
        x=x-alpha*gradf(x)
        n=n+1
        xlist.append(x)
    return x,n,np.array(xlist)
```

```
al0 = 0.4
x0 = np.array([1,1])
xopt, nopt, xoptlist=gradient_optimal(gradf,gprime,x0,al0,eps_Newton=1e-12)
xcst, ncst, xcstlist = gradient_descent(gradf, x0, al0)

plt.figure(figsize=[10,8])
plt.title("Lignes de niveau de $f$")
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.contourf(xx,yy,zz)
plt.plot(xoptlist[:,0],xoptlist[:,1],"r+-",
         label="grad. opt. n={}".format(nopt))
plt.plot(xcstlist[:,0],xcstlist[:,1],"gx-",
         label="grad. cst. n={}".format(ncst))
plt.colorbar()
plt.legend()
plt.show()
```

Exercice 6. (pour continuer) Exemple d'application de la méthode de Newton.

GPS is a system that provides the position of a receiver from signals emitted by satellites (placed in orbits located approximately 20,000 km from the center of the Earth). The principle is as follows, we measure τ_1 a time taken by a signal emitted by a satellite S_1 to reach the receiver. Knowing that the signal propagates at light speed $c = 2.9979\,10^8$ m/s we deduce a distance $d_1 = c\tau_1$. The receiver must therefore be on a sphere centered on the satellite S_1 and of radius d_1. In principle, with 3 satellites we could get the position of the receiver. In practice this is not possible, a problem comes from the synchronization of the clocks of the satellites and the receiver that is not perfect. Indeed, we measure "pseudo distances" :

$$m = d + c\Delta t$$

where Δt is the time shift between the clock of the receiver and the satellites, and it is an unknown quantity. We therefore have 4 unknowns : the coordinates of the receiver (x, y, z) and the time shift Δt. In a fixed reference, with the center of the Earth as origin, we want to solve the following :

$$m_i = \sqrt{(X_i - x)^2 + (Y_i - y)^2 + (Z_i - z)^2} + c\Delta t$$

with $v = (x, y, z, w = c\Delta t)$ the unknown, and (X_i, Y_i, Z_i, m_i) are the information from satellite i ((X_i, Y_i, Z_i) are the coordinate of satellite and m_i the measured "pseudo distance"). We need signals from 4 satellites. Let $F : \mathbb{R}^4 \to \mathbb{R}^4$ be defined by :

$$F_i(v) = \sqrt{(X_i - x)^2 + (Y_i - y)^2 + (Z_i - z)^2} + w - m_i,\ i \in \{1, 2, 3, 4\}$$

We want to solve the equation $F(v) = 0$ and we have the following measurements form 4 different satellites (available in `MesuresSatellites` file in Moodle).

Le GPS est un système permettant de donner la position d'un récepteur à partir de signaux émis par des satellites (placés sur des orbites situées à environ 20 000 km du centre de la Terre). Le principe est le suivant, on mesure un temps τ_1 mis par un signal émis par un satellite S_1 pour parvenir au récepteur. Sachant que le signal se propage à la vitesse de la lumière $c = 2.9979\,10^8$ m/s on en déduit une distance $d_1 = c\tau_1$. Le récepteur doit donc se trouver sur une sphère centrée sur le satellite S_1 et de rayon d_1. En principe, avec 3 satellites nous pourrions obtenir la position du récepteur. En pratique ce n'est pas possible, il y a un problème qui vient de la synchronisation des horloges des satellites et du récepteur qui n'est pas parfaite. On mesure en effet des "pseudo distances" :

$$m = d + c\Delta t$$

où Δt est le décalage entre l'horloge du récepteur et des satellites, c'est une quantité inconnue. Nous avons donc 4 inconnues : les coordonnées du récepteur (x, y, z) et le décalage Δt.

Dans un repère fixe dont l'origine est le centre de la Terre, nous cherchons à résoudre le problème suivant :

$$m_i = \sqrt{(X_i - x)^2 + (Y_i - y)^2 + (Z_i - z)^2} + c\Delta t$$

où, $v = (x, y, z, w = c\Delta t)$ sont les inconnues, et (X_i, Y_i, Z_i, m_i) sont les informations transmisent par le satellite i ((X_i, Y_i, Z_i) sont les coordonnées du satellite et m_i la "pseudo distance" mesurée). Nous avons donc besoin des signaux de 4 satellites. Soit $F : \mathbb{R}^4 \to \mathbb{R}^4$ définie par :

$$F_i(v) = \sqrt{(X_i - x)^2 + (Y_i - y)^2 + (Z_i - z)^2} + w - m_i,\ i \in \{1, 2, 3, 4\}$$

Nous cherchons à résoudre le problème $F(v) = 0$, et nous avons les mesures suivantes de 4 satellites différents (disponible dans le fichier `MesuresSatellites` sur Moodle).

```
# in meters
#position satellite 1 :
pos1 = [5000000, 3632713, 19021130]
m1 = 3917263658 # pseudo-distance satellite 1
#position satellite 2 :
pos2 = [-5000000, 15388418, 11755705]
m2 = 3917265503# pseudo-distance satellite 2
#position satellite 3 :
pos3 = [11180340, 3632713, -16180340]
m3 = 3917273967# pseudo-distance satellite 3
#position satellite 4 :
pos4 = [9510565, 6909830, 16180339]
m4 = 3917263997# pseudo-distance satellite 4
#definition of arrays Xi, Yi, Zi, mi
Xi = np.array([pos1[0], pos2[0], pos3[0], pos4[0]])
Yi = np.array([pos1[1], pos2[1], pos3[1], pos4[1]])
Zi = np.array([pos1[2], pos2[2], pos3[2], pos4[2]])
mi = np.array([m1, m2, m3, m4])
```

1. Implémenter la fonction F en complétant la fonction suivante :

```
def F(v) :
    x,y,z,w = v
    F = np.zeros(4)
    #to complete
    #
    return(F)
```

```
def dist(v, X, Y, Z) :
    x,y,z,w = v
    return( np.sqrt((X-x)**2 + (Y-y)**2 + (Z-z)**2) )

def F(v) :
    x,y,z,w = v
    F = np.zeros(4)
    for i in range(4) :
        F[i] = dist(v, Xi[i], Yi[i], Zi[i]) + w - mi[i]
    return(F)
```

2. Calculer la jacobienne de F et l'implémenter dans une fonction `JF` en complétant la fonction suivante :

```
def JF(v) :
   x, y, z, w = v
   J = np.zeros((4, 4))
   # to complete
   #
   return(J)
```

```
def JF(v) :
    x,y,z,w = v
    J = np.zeros((4, 4))
    for j in range(4) :
        d = dist(v, Xi[j], Yi[j], Zi[j])
        J[j, :] = [ -(Xi[j]-x)/d ,-(Yi[j]-y)/d, -(Zi[j]-z)/d, 1.]
    return(J)
```

3. Implémenter la résolution du problème non linéaire avec une méthode de Newton en complétant la fonction suivante :

```
def Newton(F, JF, v0, Tol, IterMax):
   #to complete
   #
   return(v, S, cvg, it)
```

avec

— `F` et `JF` les fonctions implémentées dans les questions précédentes

— `v0` l'initialisation de l'algorithme de Newton

— `Tol` et `IterMax` la tolérance et le nombre d'itérations maximum autorisé dans l'algorithme de Newton
— `v` le vecteur solution
— `S` la suite des itérés
— `cvg` un booléen indiquant si l'algorithme a convergé
— `it` le nombre d'itérations pour atteindre la convergence

```
def Newton(F, JF, v0, Tol, IterMax):

    #methode de Newton pour resoudre F(x)=0
    #F : R^n ---> R^n
    #JF : jacobienne de F
    #v0 dans R^n
    #sortie : v, S, cvg, Niter

    v=v0
    Fv=F(v)
    JFv=JF(v)
    z=solve(JFv,Fv)#dF(vk)^-1 . F(vk)
    S = [v0]
    it=0
    while  norm(z) > Tol and it <IterMax:
        v = v - z
        S.append(v)
        Fv, JFv = F(v, mesures), JF(v, mesures)
        z = solve(JFv,Fv)
        it+=1

    cvg = norm(z) <= Tol
    return v,np.array(S),cvg,it
```

4. Donner une estimation de v, pour une tolérance de 10^{-4} et $v0 = [0, 0, 0, 0]$.

```
v0 = np.array([0, 0, 0, 0])

v, S, cvg, it = Newton(F, JF, v0, 1.e-4, 100)
print(cvg)
print("position et decalage c*delta t : ", v)
```

5. Calculer le décalage d'horloge Δt (sachant que la vitesse de la lumière est $c = 2.9979\,10^8$ m/s).

```
c = 2.9979e8
print("decalage calculer en seconde: ", v[-1]/c)
```

Exercice 7. (pour continuer) minimisation sous contrainte

We considere the following function

$$F : (x_1, x_2) \in \mathbb{R}^2 \longmapsto ax_1^2 + bx_2^2 + cx_1x_2 + dx_1 + ex_2 + f$$

where a, b, c, d, e, f are in $\mathbb{R}$.

1. Write a code that computes, with fixed step gradient method, the solution (if it exist) of the minimization problem

$$\min_{(x_1,x_2)\in\mathbb{R}^2} F(x_1, x_2).$$

Print the obtained solution and the value of F at this point, as well as the number of iterations.
Application : $(a, b, c, d, e, f) = (3.56, 1, -3.2, -5, 0, 9.39)$,
starting point $(0, 0)$

```
def F1(X):
    x1, x2 = X
    a, b, c, d, e, f = 2.96, 1, -2.8, -5.0, 0.0, 9.39

    return(a*x1**2 + b*x2**2 + c*x1*x2 + d*x1 + e*x2 + f)

def GF(X) :
    x1, x2 = X
    a, b, c, d, e, f = 2.96, 1, -2.8, -5.0, 0.0, 9.39
    dx1 = 2*a*x1 + c*x2 + d
    dx2 = 2*b*x2 + c*x1 + e
    return(np.array([dx1, dx2]))

#meth grad
def GradientSimple(GradF,x0,rho,Tol,IterMax):
    x=x0
    gradfx=GradF(x)
    it=0
    while  norm(gradfx) > Tol and it <IterMax:
        x=x-rho*gradfx
        gradfx=GradF(x)
        it+=1
    cvg = norm(gradfx) <= Tol
    return x,cvg,it

X = GradientSimple(GF, np.array([0,0]), 0.005, 1e-4, 1000)
print("solution grad: ", X[0])
print("F(X) : ", F1(X[0]))
print("nombre it: ", X[-1])
```

2. We now want to take into account a constraint :

$$\min_{(x_1,x_2)\in\Omega} F(x_1, x_2)$$

where $\Omega \subset \mathbb{R}^2$. We propose to take care of the constraint as follow : starting from x_k, compute x_{k+1} by en projecting on Ω the array from a single step obtained with gradient method apply to x_k. We will consider the case where Ω is rectangular, which simplifies the computation of the (orthogonal) projection on Ω. Indeed, if

$$\Omega = \{(x_1, x_2), \quad a_1 \leqslant x_1 \leqslant b_1 \text{ et } a_2 \leqslant x_2 \leqslant b_2\},$$

we have

$$P_\Omega : (x_1, x_2) \in \mathbb{R}^2 \longmapsto (y_1, y_2) \in \Omega$$

where $y_i = \min(\max(a_i, x_i), b_i)$.
We can then fixed $(a_1, b_1, a_2, b_2) = (1, 4, 2, 3)$ for the rest of the exercice.

(a) Write a function that computes the projection on Ω.

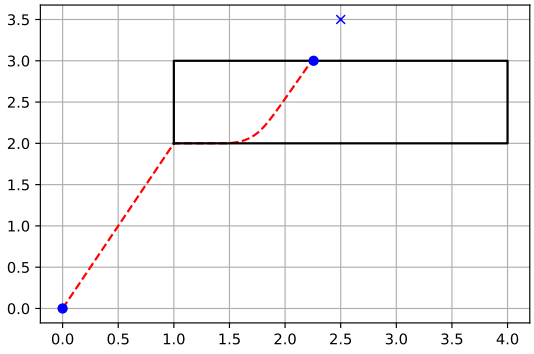
```
def proj(X) :
    a1, b1, a2, b2 = 1, 4, 2, 3
    x1, x2 = X
    y1 = min(max(a1, x1), b1)
    y2 = min(max(a2, x2), b2)
    return(np.array([y1, y2]))
```

(b) Solve the constrained problem. Print the obtained solution starting from $(0, 0)$ and the value of F.

```
def GradientContrainte(GradF,x0,rho,Tol,IterMax):
    x=x0
    gradfx=GradF(x)
    it=0
    S = [x0]
    while  norm(gradfx) > Tol and it <IterMax:
        x=proj(x-rho*gradfx)
        gradfx=GradF(x)
        it+=1
        S.append(x)
    cvg = norm(gradfx) <= Tol
    return x,np.array(S),cvg,it

x, S, cc, i = GradientContrainte(GF,np.array([0,0]),0.05,1e-4,1000)
print("x = ", x)
print("F(x) = ", F1(x))
```

(c) Draw a similar figure to the one below, where we display the obstacle, the solution without constraint (with signe “x”), the constrained minimization solution (with signe “•”), as well as a “path” starting from the initial point $(0, 0)$ and reaching the solution, meaning the sequence of points produced by the algorithm (showed in dashed line).




```
fig, ax = plt.subplots()
ax.add_patch(Rectangle((1,2),3,1, color = 'c', alpha = '0.2'))
ax.plot(S[:, 0], S[:, 1], '.', color = 'r')
ax.plot(X[0][0], X[0][1], 'X')
```