Compte-rendu TP3 Maths-info : Optimisation pour l'ingénieur

Retour en dimension 1

(1)

On considère la fonction $f(x) = x^2 - \frac{1}{4}x^4$. On considère les points x_k correspondant à une descente de gradient à pas fixe α pour la fonction f. On programme numériquement la suite x_k pour des valeurs de α et de x_0 où la convergence est linéaire. On fixe x_0 = 0.5.

Tout d'abord, on programme la fonction f puis son gradient. Ensuite, on utilise la méthode du gradient à pas fixe afin de trouver le point de convergence de f.

Graphiquement, on remarque que sur l'intervalle [-2.1,2.1], le minimum local de la fonction est atteint en 0 et vaut 0.

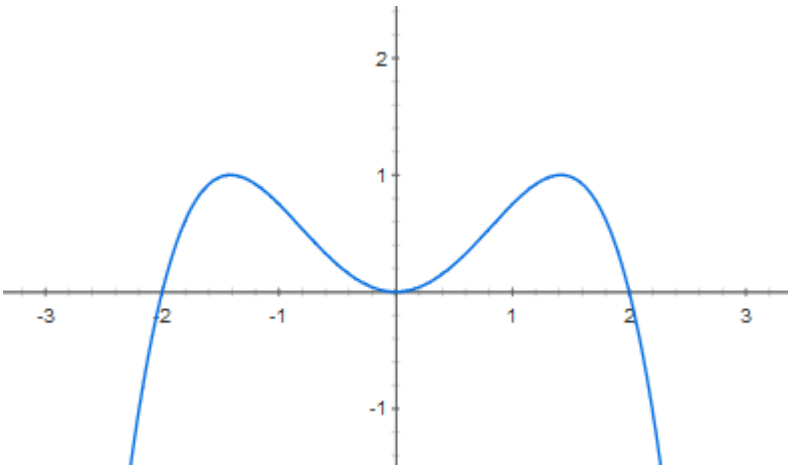
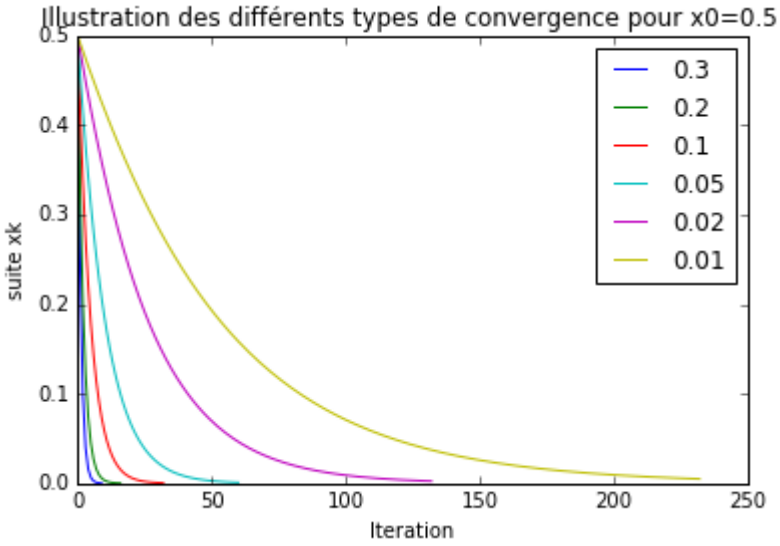


Figure 1 : Représentation graphique de la fonction f

```
def f(x): # définition de f
    return (x**2 - (1/4)*(x**4))


def Grad_f(x): # définition du gradient de f
    return (2*x - x**3)


def Grad_fixe_f(x,alpha):
    #Approximation de f par la méthode du gradient à pas fixe
    k = 0
    kmax=1000
    epsilon=10**(-4)
    X = [x]
    K = [k]
    while np.abs(- alpha*(Grad_f(x)))>epsilon and k < kmax:
        x= x - alpha*(Grad_f(x))
        k = k+1
        K.append(k)
        X.append(x)
    return (X,K)
```

Dans ce compte-rendu, nous ne détaillerons pas les lignes de code écrites pour les affichages graphiques. Nous nous en tiendrons aux résultats mais pour plus de détails : vous pouvez consulter le code fourni.

Les différents types de convergence obtenus sont illustrés graphiquement sur la figure ci-dessous.



Pour différentes valeurs de alpha, on converge bien vers 0. La convergence est donc vérifiée. Nous observons qu'un pas trop fin diminue grandement la vitesse de

convergence. Il faut donc jouer entre précision et rapidité. De plus, un pas trop grand ne permettra plus de converger. Par ailleurs, comme on peut le déduire à la vue du graphique, il faut que la valeur initiale : x0 soit dans l'intervalle $]-\sqrt{2},\sqrt{2}[$.

(2)

Soit la fonction $f(x)=\frac{x^2}{1+x^2}$. On considère les points x_k correspondant à la méthode de descente de gradient à pas fixe α. On programme la suite x_k lorsque $x_0=3$ et α = 1/3 et on compare cette méthode à celle de la section dorée (gradient à pas optimal).

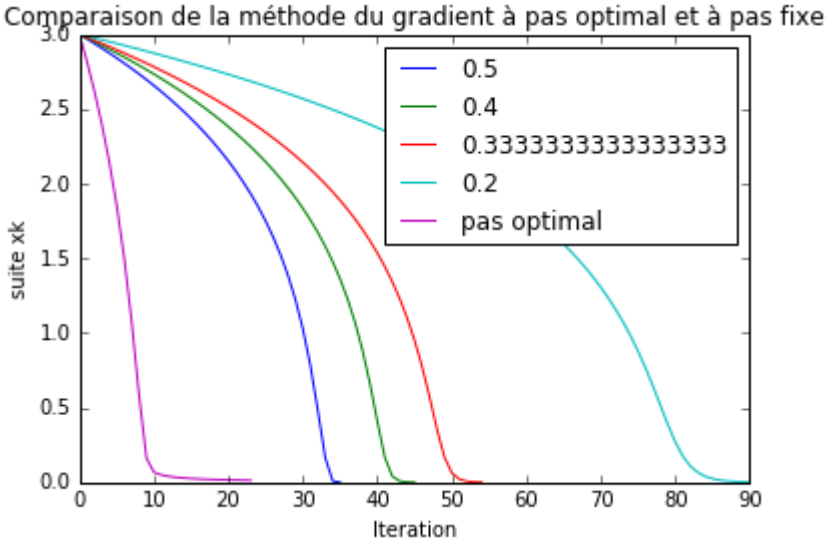
De la même façon que dans la première question, on programme notre nouvelle fonction notée f_2. Puis, on applique la méthode du gradient à pas fixe.

```
def f_2(x): # définition de la fonction f de la question 2 nommée f_2
    return (x**2/(1+x**2))

def Grad_f_2(x): # définition du gradient de la fonction
    return (2*x)/((1+x**2)**2)


def Grad_fixe_f_2(x,alpha):
    #Approximation de f_2 par la méthode du gradient à pas fixe
    k = 0
    kmax=1000
    epsilon=10**(-3)
    X = [x]
    K = [k]
    while np.abs(- alpha*(Grad_f_2(x)))>epsilon and k < kmax:
        x= x - alpha*(Grad_f_2(x))
        k = k+1
        K.append(k)
        X.append(x)
    return (X,K)
```

Ensuite, on programme la méthode de la section dorée qui nous permet dans l'algorithme du gradient à pas optimal de trouver à chaque itération le pas optimal et d'ainsi optimiser l'approche de la convergence.

```
def SectionDoree(a0,b0,n_max,f):
    t = (1+np.sqrt(5))/2
    a,b = a0,b0
    for i in range(n_max):
        A = a + (b-a)/(t**2)
        B = a + (b-a)/t
        if f(A) > f(B) :
            b = B
        elif f(A) < f(B):
            a = A
        else :
            a,b = A,B
    return (a+b)/2

def GradPasOptimal(X0):
    epsilon=10**(-3)
    kmax=100
    X = [X0]
    x = X0
    dx = Grad_f_2(x)
    k=0
    K = [k]
    while k < kmax and np.abs(x**2+dx**2) >=epsilon:
        b = SectionDoree(x,dx,100,f_2)
        x=x-b*dx
        dx = Grad_f_2(x)
        k+=1
        K.append(k)
        X.append(x)
    return (X,K)
```

On procède alors à l'affichage de la convergence de la fonction f (notée f_2 dans le programme) par la méthode du gradient à pas fixe et par la méthode du gradient à pas optimal. On ne détaillera pas le code permettant d'obtenir le graphique comme précisé précédemment.



Le nombre d'itérations pour la méthode du gradient à pas optimal est ici de 23.

La méthode du gradient à pas optimal assure une convergence plus rapide que la méthode du gradient à pas fixe lorsque le pas est inférieur à 0,5.
Au-dessus de 0,5 la méthode du gradient à pas fixe est donc plus performante jusqu'à une certaine limite ou il n'y a plus convergence. Il est donc impossible de généraliser quant à la méthode la plus performante. En effet, la méthode du gradient à pas optimal est très rapide durant les premières itérations mais ralentit beaucoup à l'approche de la convergence.

Regarder au préalable le comportement de la suite qui dépend de la fonction, de la méthode et du pas choisi, est donc intéressant pour savoir quelle méthode permet de converger le plus rapidement et est donc la plus performante.

(3)

On applique maintenant la méthode du gradient à pas optimal sur la fonction $f_a(x_0,x_1)=1-\frac{1}{1+ax_0^2+x_1^2}, a>0$ et on affiche les points obtenus sur un graphique.

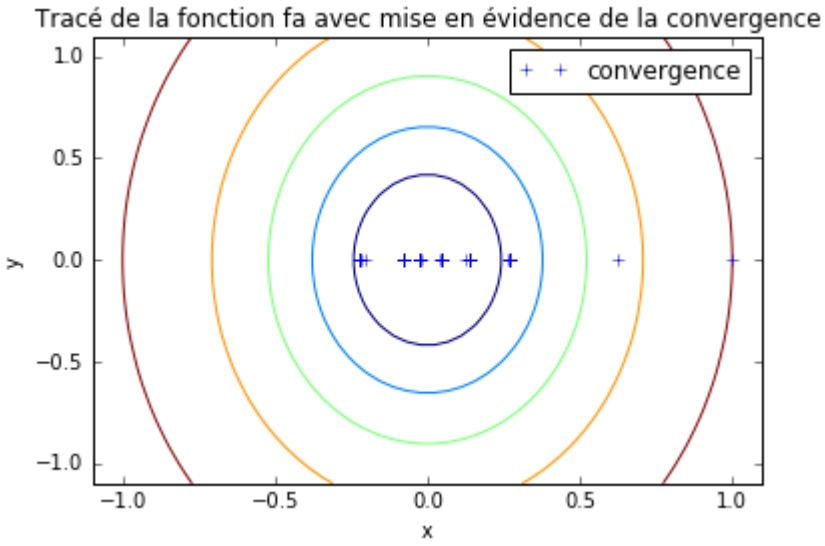
On pose a=3. On définit alors la fonction, son gradient et on réalise une deuxième fonction pouvant réaliser la méthode de la section dorée dans le cas de la dimension 2 et non dimension 1 comme dans le cas précédent.

```
a=3

def fa(X): # X estun vecteur
    if a > 0:
        return 1-1/(1+a*(X[0]**2)+X[1]**2)

def grad_fa(X):
    return( 2*a*X[0]/(1+a*X[0]**2+X[1]**2)**2 , 2*X[1]/(1+a*X[0]**2+X[1]**2)**2


def SectionDoree_2(a0,b0,n_max,f):
    t = (1+np.sqrt(5))/2
    a,b = a0,b0
    A=[0,0]
    B=[0,0]
    for i in range(n_max):
        A[0] = a[0] + (b[0]-a[0])/(t**2)
        A[1] = a[0] + (b[0]-a[0])/(t**2)

        B[0]= a[0] + (b[0]-a[0])/t
        B[1]= a[1] + (b[1]-a[1])/t

        if f(A) > f(B) :
            b = B
        elif f(A) < f(B):
            a = A
        else :
            a,b = A,B

    return (mp.sqrt(a[0]**2+a[1]**2)+mp.sqrt(b[0]**2+b[1]**2))/2
```

Puis, on programme une deuxième fonction de la méthode du gradient optimal s'adaptant à la fonction à deux variables de cette question.

```
def GradPasOptimal_2(X0):
    epsilon=10**(-3)
    kmax=100
    X = [X0]
    x1 = [X0[0]]
    y1 = [X0[1]]
    [x,y] = X0
    F = [fa([x,y])]

    [dx,dy] = grad_fa([x,y])
    Dx=[dx,dy]
    k=1

    while k < kmax and mp.sqrt(dx**2+dy**2) >=epsilon:

        b = SectionDoree_2(X[-1],Dx,100,fa)

        x,y=x-b*dx,y-b*dy
        [dx,dy] = grad_fa([x,y])
        k+=1
        F.append(fa([x,y]))
        X.append([x,y])
        x1.append(x)
        y1.append(y)

    return (F,X,k,x1,y1,Dx)
```

Ensuite, on peut donc afficher les points obtenus sur un graphique :



Il y a bien convergence vers le point (0,0) et on remarque le principe du pas optimal qui permet de s'approcher rapidement du point recherché. Puis, après dépassement de ce dernier, il est affiné afin d'apporter de la précision.

Méthode du gradient projeté

Dans cette deuxième partie, nous allons appliquer la méthode du gradient projeté vu dans le cours à l'exemple suivant :

On considère la fonctionnelle J définie sur $\square^2$ par : $J(x,y) = 2x^2 + 3xy + 2y^2$

On appelle Q le cadrant défini par :

$$Q = \{x \leq -1/2, y \leq -1/2\}$$

La solution du problème de minimisation de J sans contrainte est (0;0)

Tout d'abord, comme dans chaque exercice, on définit la fonction ainsi que son gradient.

```
def J (x,y):
    return 2*(x**2)+3*x*y+2*(y**2)

def dJx(x, y):
    return 4*x + 3*y

def dJy(x, y):
    return 3*x + 4*y

def gradientJ(x, y):

    return np.array([dJx(x, y), dJy(x, y)])
```

Ensuite, on définit la fonction de projection du vecteur X. Ici, la zone de projection (espace des contraintes) est définie par les conditions x <= x_max et y <= y_max. Avec x_max = y_max= -0.5.

```
def proj(X):

    return np.array([min(x_max, X[0]), min(y_max, X[1])])
    # Pour chaque vecteur, on lui associe le point le plus proche appartenant
    # à l'espace des contraintes.
```

Puis, on définit la méthode du gradient projeté (pas fixe).

```
def Gradiant_Pas_fixe_Projete(X0, pas):

    n_max=1000
    epsilon=1e-8
    x, y = X0
    i = 0

    d = -gradientJ(x, y)
    X1 = proj(X0 + pas*d)


    while (i < n_max) and (np.abs((X1[0]-X0[0])**2+(X1[1]-X0[1])**2) >= epsilon):

        d = -gradientJ(X1[0], X1[1])
        X0 = X1
        X1 = proj(X1 + pas*d)

    return X1

X = np.array([1,0])
alpha = 0.1
x_max = -0.5
y_max = -0.5
```

Finalement, on obtient :