Méthode des moindres carrés

1. Introduction

On dispose de N données expérimentales (x_i, y_i). Les valeurs x_i sont certaines alors que y_i est une variable aléatoire dont on connaît l'écart-type, noté σ_i.

On recherche une fonction $f(x|a_0, a_1)$ comportant deux paramètres (a_0, a_1) qui représente au mieux ces données expérimentales. Dans cet exposé, on se limite à la recherche d'une fonction à deux paramètres mais les algorithmes seront généralisables à une fonction comportant un nombre quelconque de paramètres.

La méthode des moindres carrés consiste à recherche les paramètres qui minimisent la fonction erreur définie par :

$$e(a_0, a_1) = \sum_{i=0}^{N-1} \left(\frac{y_i - f(x_i|a_0, a_1)}{\sigma_i} \right)^2 \tag{1}$$

Sans donner le fondement statistique de cette méthode, on remarque simplement que l'écart au carré est pondéré par l'inverse de la variance : une donnée très incertaine a donc un poids plus faible dans la somme.

Nous allons tout d'abord traiter le cas d'une fonction qui dépend linéairement des paramètres, qui conduit à l'équation normale, dont la résolution relève de l'algèbre linéaire. Dans un second temps, nous aborderons le cas d'une fonction non linéaire des paramètres, un problème beaucoup plus difficile d'optimisation dans un espace de dimension 2 (autant que de paramètres), qui peut se résoudre par la méthode itérative du gradient.

Remarque : la méthode des moindres carrés est une méthode élémentaire d'apprentissage. Si un grand nombre de données correspondant au même phénomène est traité, la connaissance des paramètres (a_0, a_1) permet de faire une prévision sur une nouvelle donnée x_n en appliquant : $y_n = f(x_n|a_0, a_1)$.

2. Problème linéaire

2.a. Équation normale

Lorsque la fonction recherchée dépend linéairement de ses paramètres, elle a la forme générale suivante :

$$f(x|a_0, a_1) = a_0 f_0(x) + a_1 f_1(x) \tag{2}$$

Les fonctions f_0 et f_1 sont quelconques. La fonction erreur s'écrit :

$$e(a_0, a_1) = \sum_{i=0}^{N-1} \left(\frac{y_i - a_0 f_0(x_i) - a_1 f_1(x_i)}{\sigma_i} \right)^2 \tag{3}$$

Dans le cas d'un problème linéaire, le minimum de la fonction erreur est obtenu sans équivoque par la condition d'annulation de ses dérivées partielles (c'est-à-dire de son gradient) :

$$\frac{\partial e(a_0, a_1)}{\partial a_0} = 0 \tag{4}$$
$$\frac{\partial e(a_0, a_1)}{\partial a_1} = 0 \tag{5}$$

En explicitant les dérivées partielles :

$$\sum_{i=0}^{N-1} \frac{2}{\sigma_i^2} \left(y_i - a_0 f_0(x_i) - a_1 f_1(x_i)\right) f_0(x_i) = 0 \tag{6}$$
$$\sum_{i=0}^{N-1} \frac{2}{\sigma_i^2} \left(y_i - a_0 f_0(x_i) - a_1 f_1(x_i)\right) f_1(x_i) = 0 \tag{7}$$

Les paramètres qui rendent l'erreur minimale sont donc solutions du système linéaire suivant :

$$a_0 \sum_{i=0}^{N-1} \frac{f_0(x_i) f_0(x_i)}{\sigma_i^2} + a_1 \sum_{i=0}^{N-1} \frac{f_1(x_i) f_0(x_i)}{\sigma_i^2} = \sum_{i=0}^{N-1} \frac{y_i f_0(x_i)}{\sigma_i^2} \tag{8}$$
$$a_0 \sum_{i=0}^{N-1} \frac{f_0(x_i) f_1(x_i)}{\sigma_i^2} + a_1 \sum_{i=0}^{N-1} \frac{f_1(x_i) f_1(x_i)}{\sigma_i^2} = \sum_{i=0}^{N-1} \frac{y_i f_1(x_i)}{\sigma_i^2} \tag{9}$$

Une écriture matricielle permettra de généraliser à un nombre quelconque de paramètres. On introduit pour cela la matrice :

$$A = \begin{pmatrix} \frac{f_0(x_0)}{\sigma_0} & \frac{f_1(x_0)}{\sigma_0} \\ \frac{f_0(x_1)}{\sigma_1} & \frac{f_1(x_1)}{\sigma_1} \\ \frac{f_0(x_2)}{\sigma_2} & \frac{f_1(x_2)}{\sigma_2} \\ \cdots & \cdots \end{pmatrix} \tag{10}$$

Cette matrice comporte N lignes (nombre de données) et autant de colonnes que de paramètres.

On définit aussi la matrice colonne suivante :

$$b = \begin{pmatrix} \sum_{i=0}^{N-1} \frac{y_i f_0(x_i)}{\sigma_i^2} \\ \sum_{i=0}^{N-1} \frac{y_i f_1(x_i)}{\sigma_i^2} \end{pmatrix} \tag{11}$$

qui comporte autant de lignes que de paramètres. Le système linéaire à résoudre s'écrit alors :

$$(A^T \cdot A) \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = b \tag{12}$$

La matrice inverse

$$C = (A^T \cdot A)^{-1} \tag{13}$$

est également recherchée car elle donne les variances des coefficients et les covariances. Les éléments diagonaux sont les variances des coefficients (évaluées à partir des variances de y_i) :

$$\sigma^2(a_k) = C_{kk} \tag{14}$$

Ces variances fournissent une information sur la fiabilité à attribuer aux coefficients calculés.

La résolution du système devra donc se faire avec une méthode qui fournit aussi la matrice inverse, par exemple la méthode d'élimination de Gauss-Jordan.

2.b. Simulation d'une expérience

```
import numpy
from matplotlib.pyplot import *
from mpl_toolkits.mplot3d import Axes3D
import random
import numpy.linalg
```

Pour tester la méthode des moindres carrés, on définit une fonction $f(x|a_0, a_1)$ et on génère des données aléatoires correspondant à cette fonction. Pour simplifier, les N valeurs x_i sont échantillonnées régulièrement sur l'intervalle $[0, 1]$. y_i est une variable aléatoire continue. On suppose que sa densité de probabilité est la loi normale (ou loi de Gauss) avec une espérance $f(x_i|a_0, a_1)$ et un écart type σ_i. Remarque : la loi normale pour les phénomènes expérimentaux est très souvent utilisée mais il ne faut pas oublier que certains phénomènes ne se plient pas du tout à cette loi.

Pour les tests, on choisit la fonction suivante (régression linéaire) :

$$f(x|a_0, a_1) = a_0 + a_1 x \tag{15}$$

On définit les deux fonctions f_0, f_1 sous forme de fonctions python, ce qui permettra de traiter d'autres exemples :

```
def f0(x):
    return 1

def f1(x):
    return x
```

Pour générer des nombres réels (à virgule flottante) avec une distribution normale, on utilise la fonction `random.gauss(mu,sigma)`.

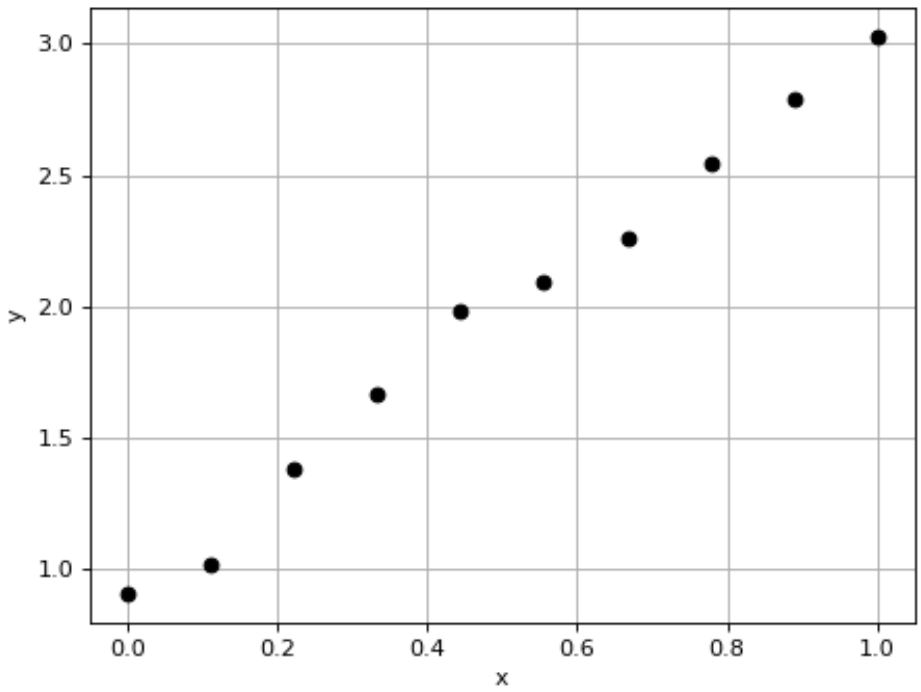
La fonction `generation(a0,a1,f0,f1,N,s)` génère aléatoirement les données pour des paramètres et des fonctions donnés. L'écart-type est supposé identique pour toutes les données ; il est donné par l'agument `s`. La fonction renvoie `(x,y,sigma)`, c'est-à-dire le tableau des valeurs x_i, le tableau des valeurs y_i et le tableau des valeurs σ_i.

```
def generation(a0,a1,f0,f1,N,s):
    x=numpy.linspace(0,1,N)
    sigma=numpy.ones(N)*s
```

```
    y=numpy.zeros(N)
    for i in range(N):
        yi=a0*f0(x[i])+a1*f1(x[i])
        y[i] = random.gauss(yi,sigma[i])
    return (x,y,sigma)
```

Voici un exemple de génération, que l'on utilisera par la suite :

```
N=10
a0=1
a1=2
s=0.1
(x,y,sigma)=generation(a0,a1,f0,f1,N,s)
figure()
plot(x,y,"ko")
xlabel("x")
ylabel("y")
grid()
```



Pour implémenter la méthode des moindres carrés, on définit les matrices A et b. La matrice transposée s'obtient par l'opérateur `A.T`. Le produit de deux matrices est obtenu avec `numpy.dot`. Pour obtenir la matrice inverse, on utilise la fonction `numpy.linalg.inv`.

La fonction `moindre_carre_lineaire(f0,f1,x,y,sigma)` implémente la méthode des moindres carrés et renvoie `(a,C)`, le tableau des valeurs des paramètres et la matrice de covariance.

```
def moindre_carre_lineaire(f0,f1,x,y,sigma):
    N=len(x)
    A=numpy.zeros((N,2))
    b=numpy.zeros(2)
    b[0] = 0
    b[1] = 0
    for i in range(N):
        A[i,0] = f0(x[i])/sigma[i]
        A[i,1] = f1(x[i])/sigma[i]
        b[0] += y[i]*f0(x[i])/sigma[i]**2
        b[1] += y[i]*f1(x[i])/sigma[i]**2
    C=numpy.linalg.inv(numpy.dot(A.T,A))
    a = numpy.dot(C,b)
    return(a,C)
```

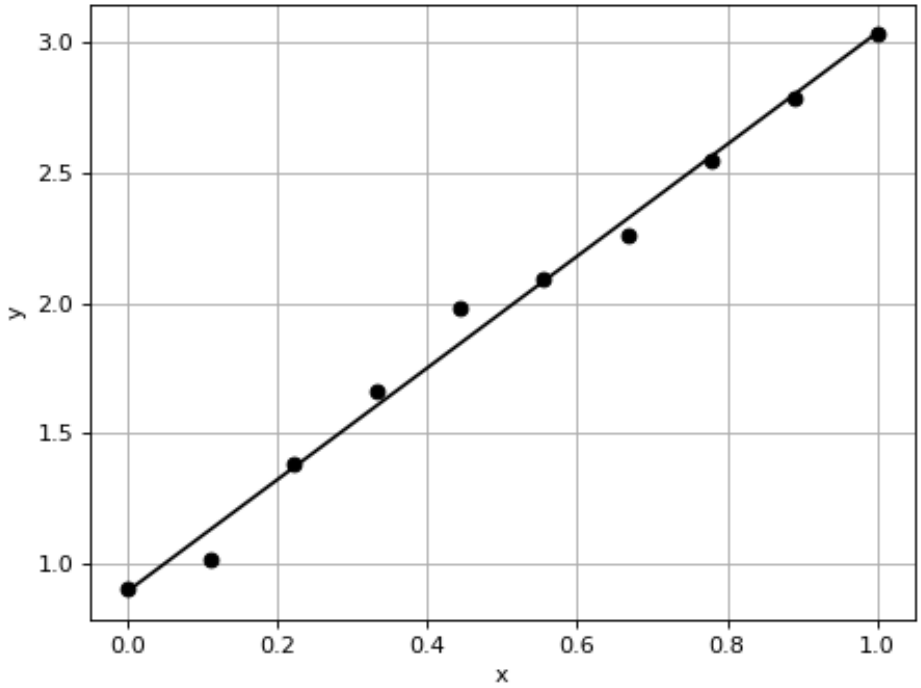
Voici l'application aux données générées précédemment :

```
(a,C)=moindre_carre_lineaire(f0,f1,x,y,sigma)


print(a)
--> array([ 0.8946197 ,  2.14281351])

print(C)
--> array([[ 0.00345455, -0.00490909],
       [-0.00490909,  0.00981818]])

list_x=numpy.linspace(0,1,500)
plot(list_x,a[0]*f0(list_x)+a[1]*f1(list_x),"k-")
xlabel("x")
ylabel("y")
```



Les écart-types des deux paramètres sont :

```
print([numpy.sqrt(C[0,0]),numpy.sqrt(C[1,1])])
--> [0.058775381364525849, 0.099086738861372425]
```

2.c. Distribution des valeurs des paramètres

Les valeurs des paramètres qui minimisent la fonction erreur dépendent bien-sûr des données expérimentales. Les écart-types évalués précédemment donnent une première indication sur la distribution de ses valeurs, mais il peut être intéressant d'évaluer la densité de probabilité $p(a_0, a_1)$ associée aux deux paramètres, par une méthode de Monte-Carlo. Il s'agit de calculer un histogramme dans un espace à deux dimensions, c'est-à-dire un tableau de M lignes et M colonnes. Il faut choisir un intervalles de valeurs de a_0 et un intervalle de valeurs de a_1. Le nombre de tirages N_t, c'est-à-dire le nombre de générations aléatoires des données suivie de l'application de la méthode des moindres carrés, doit être grand devant M^2.

La fonction `hist_coefficients(a0,a1,delta_a0,delta_a1,M,Nt,f0,f1,s)` calcule l'histogramme des coefficients. Les arguments `a0,a1,delta_a0,delta_a1` définissent les centres et les largeurs des intervalles de valeurs considérées pour les deux coefficients. La fonction renvoie `(list_a0,list_a1,h)` : le tableau des valeurs de a, celui des valeurs de b et le tableau contenant l'histogramme.

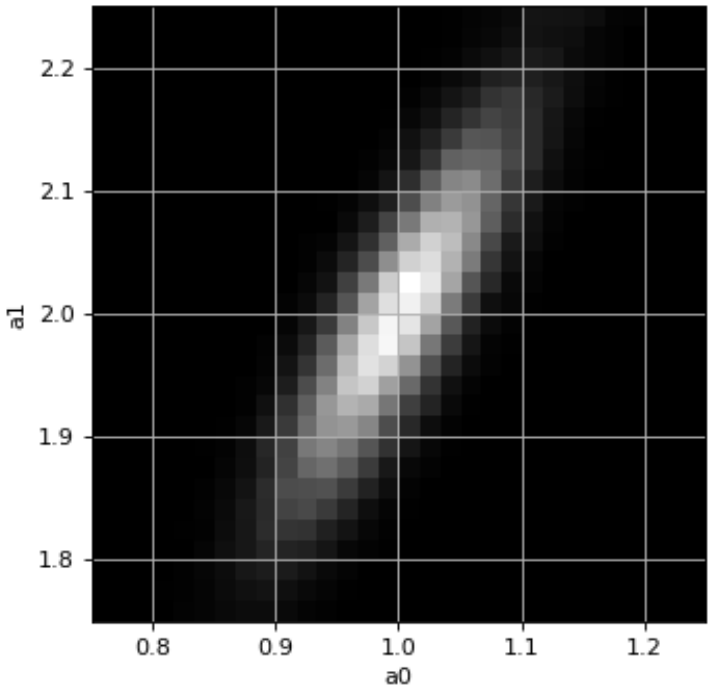
```
def hist_coefficients(a0,a1,delta_a0,delta_a1,M,Nt,f0,f1,s):
    h=numpy.zeros((M,M))
    a0_min=a0-delta_a0/2
    a1_min=a1-delta_a1/2
    da0 = delta_a0/M
```

```
    da1 = delta_a1/M
    for t in range(Nt):
        (x,y,sigma) = generation(a0,a1,f0,f1,N,s)
        (a,C)=moindre_carre_lineaire(f0,f1,x,y,sigma)
        i = int((a[0]-a0_min)/da0)
        j = int((a[1]-a1_min)/da1)
        if (i>=0) and (i<M) and (j>=0) and (j<M):
            h[j,i] += 1
    list_a0 = numpy.linspace(a0_min,a0_min+delta_a0,M)
    list_a1 = numpy.linspace(a1_min,a1_min+delta_a1,M)
    return (list_a0,list_a1,h)
```

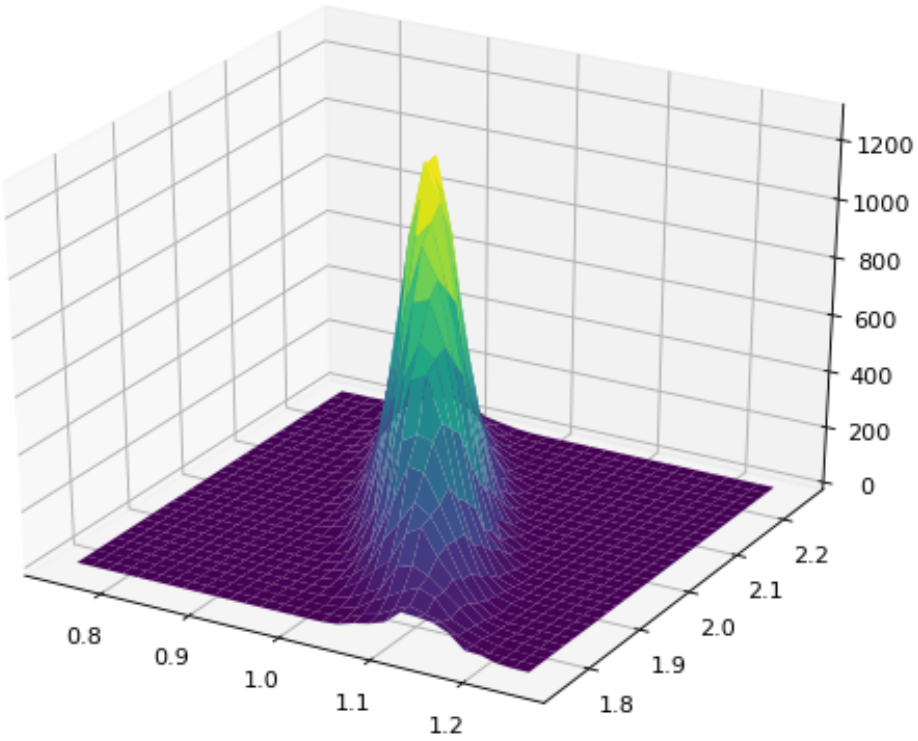
Voici une simulation complète, avec représentation de l'histogramme sous forme d'une image puis sous forme d'une surface :

```
delta_a0 = 0.5
delta_a1 = 0.5
M=30
Nt = M**2*100
(list_a0,list_a1,h) = hist_coefficients(a0,a1,delta_a0,delta_a1,M,Nt,f0,f1,s)

imshow(h,cmap=cm.gray,extent=[a0-delta_a0/2,a0+delta_a0/2,a1-delta_a1/2,a1+delta_a1/2]
xlabel("a0")
ylabel("a1")
```



```
fig=figure()
ax = Axes3D(fig)
grid_a0,grid_a1=numpy.meshgrid(list_a0,list_a1)
ax.plot_surface(grid_a0, grid_a1, h, rstride=1, cstride=1, cmap=cm.viridis)
```

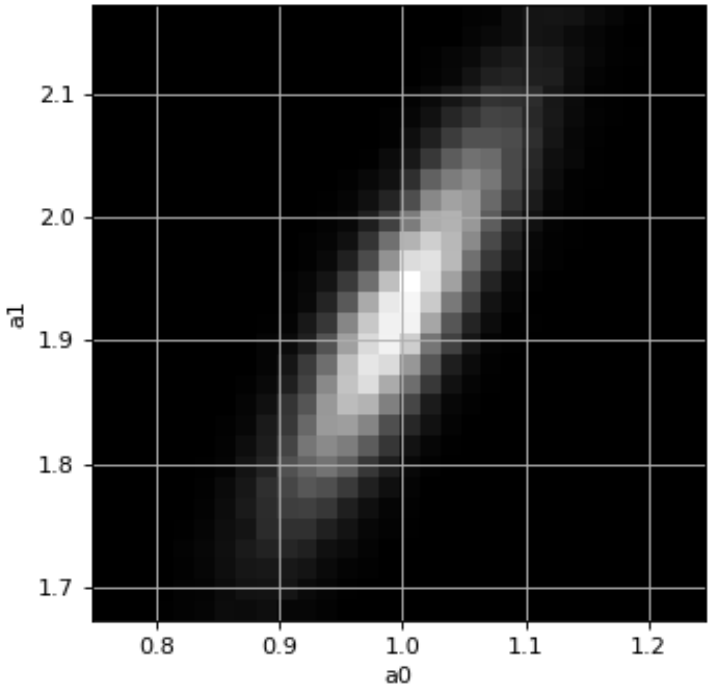


Supposons à présent que les paramètres exacts soient inconnus et que l'on dispose seulement d'une réalisation expérimentale de N échantillons. La méthode des moindres carrés fournit une estimation des paramètres, notée a_0^{exp}, a_1^{exp}. Une étude de Monte-Carlo de la distribution des valeurs des paramètres est alors menée comme précédemment, mais en faisant l'hypothèse que ces valeurs expérimentales sont les valeurs exactes.

Voici l'application de cette méthode :

```
# une expérience
(x,y,sigma)=generation(a0,a1,f0,f1,N,0.1)
# estimation des coefficients
(a,C)=moindre_carre_lineaire(f0,f1,x,y,sigma)
# estimation de la distribution des valeurs des coefficients
# utilisation des coefficients calculés pour la simulation de Monte-Carlo
a0=a[0]
a1=a[1]
delta_a0 = 0.5
delta_a1 = 0.5
M=30
Nt = M**2*100
(list_a0,list_a1,h) = hist_coefficients(a0,a1,delta_a0,delta_a1,M,Nt,f0,f1,s)
```

```
figure()
imshow(h,cmap=cm.gray,extent=[a0-delta_a0/2,a0+delta_a0/2,a1-delta_a1/2,a1+delta_a1/2
xlabel("a0")
ylabel("a1")
grid()
```



Bien sûr, la distribution obtenue est centrée sur les valeurs expérimentales des paramètres et non pas sur les valeurs exactes. Cependant, l'étalement des valeurs autour du maximum est similaire à celui obtenu plus haut avec les valeurs exactes des paramètres. Cette évaluation de la densité de probabilité $p(a, b)$ permet de déterminer une region de confiance avec un certain pourcentage, typiquement 68%, qui a la forme d'une ellipse.

3. Problème non linéaire

3.a. Algorithme de descente par le gradient

La fonction à ajuster aux données expérimentales dépend de ses paramètres de manière non linéaire. On étudiera l'exemple suivant :

$$f(x|a_0, a_1) = a_0 \cos(2\pi a_1 x) \tag{16}$$

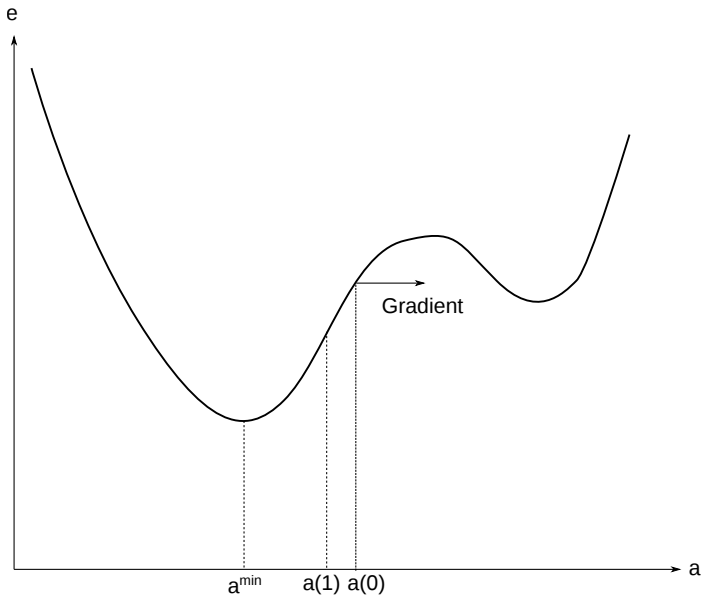
On doit déterminer les paramètres (a_0, a_1) qui minimisent la fonction erreur $e(a_0, a_1)$. Il s'agit d'un problème d'optimisation qui peut, sous certaines conditions, se résoudre avec la méthode du gradient.

Considérons le gradient de la fonction erreur :

$$\frac{\partial e}{\partial a_0} = -2\sum_{i=0}^{N-1}\frac{(y_i - f(x_i|a_0,a_1))}{\sigma_i^2}\frac{\partial f}{\partial a_0} \tag{17}$$

$$\frac{\partial e}{\partial a_1} = -2\sum_{i=0}^{N-1}\frac{(y_i - f(x_i|a_0,a_1))}{\sigma_i^2}\frac{\partial f}{\partial a_1} \tag{18}$$

Pour introduire la méthode de descente par le gradient (ou algorithme de descente du gradient), considérons la recherche du minimum d'une fonction erreur $e(a)$ à une variable.



Soit une valeur initiale du paramètre $a(0)$. Le gradient calculé pour cette valeur du paramètre est indiqué par la flèche sur la figure. On voit qu'il faut faire évoluer la valeur du paramètre dans la direction opposée au gradient pour s'approcher du minimum. On calcule donc une seconde valeur du paramètre par :

$$a(1) = a(0) - h\frac{de}{da} \tag{19}$$

où h est une constante assez petite. Le gradient est à nouveau calculé pour cette valeur du paramètre et la procédure est répétée itérativement jusqu'à convergence. Si le minimum est un point stationnaire alors la variation appliquée au paramètre est de plus en plus petite lorsqu'on s'approche du minimum. On stoppe les itérations lorsque la norme du gradient est inférieure à une petite valeur ϵ, ce qui signifie que l'on est très proche du minimum. On voit sur la figure précédente que si la valeur initiale est au voisinage d'un minimum local, la méthode converge vers celui-ci au lieu de converger vers le minimum absolu. De manière générale, c'est la difficulté principale des problèmes d'optimisation.

Dans le cas d'une fonction erreur à deux paramètres, la méthode de descente par le gradient consiste à construire une suite de valeurs $(a_0(k), a_1(k))$ déduites les unes des autres par un petit déplacement dans la direction opposée au gradient :

$$a_0(k+1) = a_0(k) - h\frac{\partial e}{\partial a_0}(a_0(k), a_1(k)) \tag{20}$$

$$a_1(k+1) = a_1(k) - h\frac{\partial e}{\partial a_1}(a_0(k), a_1(k)) \tag{21}$$

La direction opposée au gradient est la direction de la plus grande pente descendante, c'est pourquoi cette méthode est aussi appelée méthode de la plus grande pente descendante. Dans le contexte des algorithmes d'apprentissage, la constante h est appelée taux d'apprentissage. La difficulté de la mise en œuvre de cette méthode est le choix de la constante h permettant d'assurer une convergence rapide sans risquer de passer à côté du minimum. Par ailleurs, la plus grande pente n'est pas toujours le chemin le plus rapide vers le minimum.

3.b. Implémentation

On définit tout d'abord la fonction recherchée et son gradient :

```
def f(x,a0,a1):
    return a0*numpy.cos(2*numpy.pi*a1*x)

def df_da0(x,a0,a1):
    return numpy.cos(2*numpy.pi*a1*x)

def df_da1(x,a0,a1):
    return -x*a0*numpy.sin(2*numpy.pi*a1*x)
```

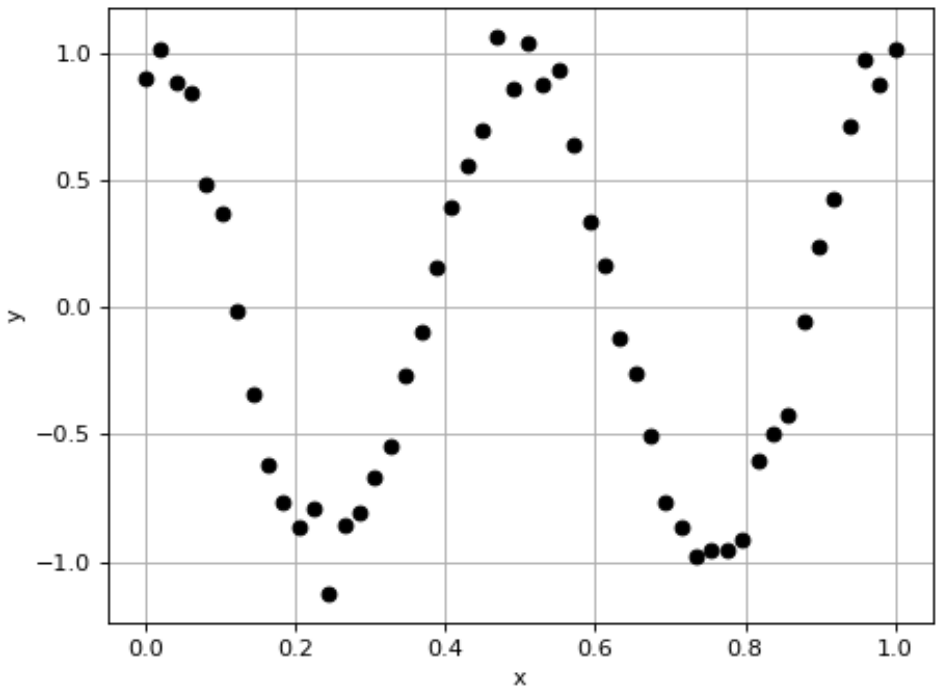
On écrit une fonction `generation(a0,a1,f,N,s)` qui génère des valeurs pour des paramètres donnés.

```
def generation(a0,a1,f,N,s):
    x=numpy.linspace(0,1,N)
    sigma=numpy.ones(N)*s
    y=numpy.zeros(N)
    for i in range(N):
        yi=f(x[i],a0,a1)
        y[i] = random.gauss(yi,sigma[i])
    return (x,y,sigma)
```

On choisit $a_0 = 1$ et $a_1 = 2$ et on génère 50 données :

```
N=50
a0=1
a1=2
s=0.1
(x,y,sigma)=generation(a0,a1,f,N,s)
figure()
```

```
plot(x,y,"ko")
xlabel("x")
ylabel("y")
grid()
```



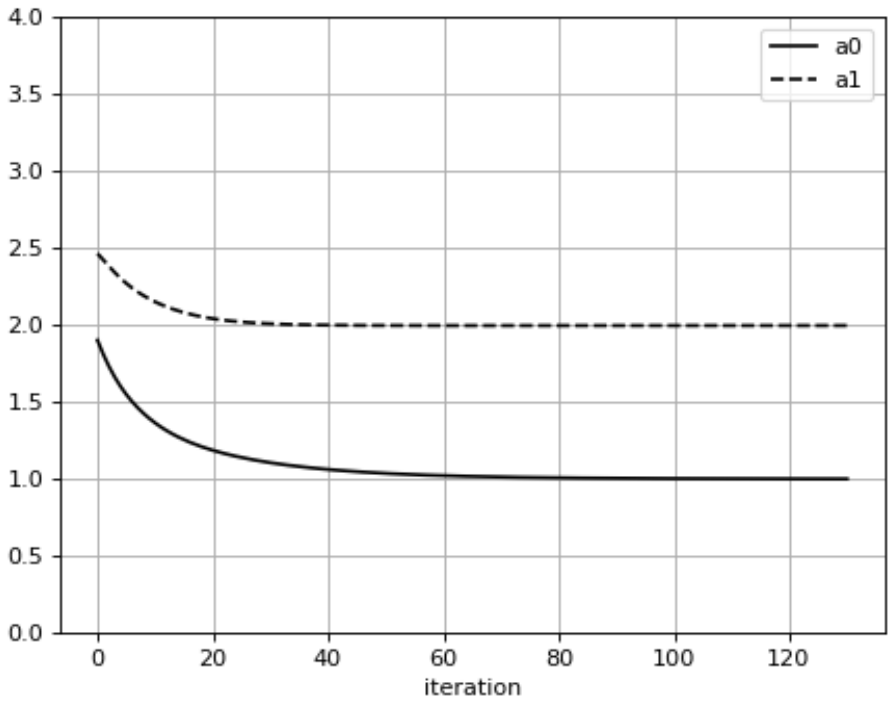
La fonction `moindre_carre_non_lineaire_gradient(f,x,y,sigma,a0i,a1i,h,eps)` effectue la recherche du minimum à partir des valeurs initiales `a0i,a1i`, pour une constante `h` donnée. La recherche est stoppée lorsque la norme du gradient est inférieure à `eps`. La fonction renvoit le tableau des valeurs successives de a_0, celui des valeurs de a_1 et celui des valeurs de l'erreur.

```
def moindre_carre_non_lineaire_gradient(f,x,y,sigma,a0i,a1i,h,eps):
    N=len(x)
    iteration = True
    a0 = a0i
    a1 = a1i
    eps2 = eps**2
    list_a00 = []
    list_a01 = []
    list_e = []
    while iteration:
        print(a0,a1)
        de_da0 = 0
        de_da1 = 0
        e = 0
        for i in range(N):
```

```
            ecart = y[i]-f(x[i],a0,a1)
            e += ecart**2/sigma[i]**2
            z = -2*ecart/sigma[i]**2
            de_da0 += z*df_da0(x[i],a0,a1)
            de_da1 += z*df_da1(x[i],a0,a1)
        a0 += - h*de_da0
        a1 += - h*de_da1
        list_a00.append(a0)
        list_a01.append(a1)
        list_e.append(e)
        if h*(de_da0**2+de_da1**2) < eps2:
            iteration = False
    return (list_a00,list_a01,list_e)
```

Voici un exemple :

```
a0i=2
a1i=2.5
(list_a00,list_a01,list_e) = moindre_carre_non_lineaire_gradient(f,x,y,sigma,a0i,a1i,
figure()
plot(list_a00,"k-",label="a0")
plot(list_a01,"k--",label="a1")
xlabel("iteration")
legend(loc="upper right")
grid()
ylim(0,4)
```



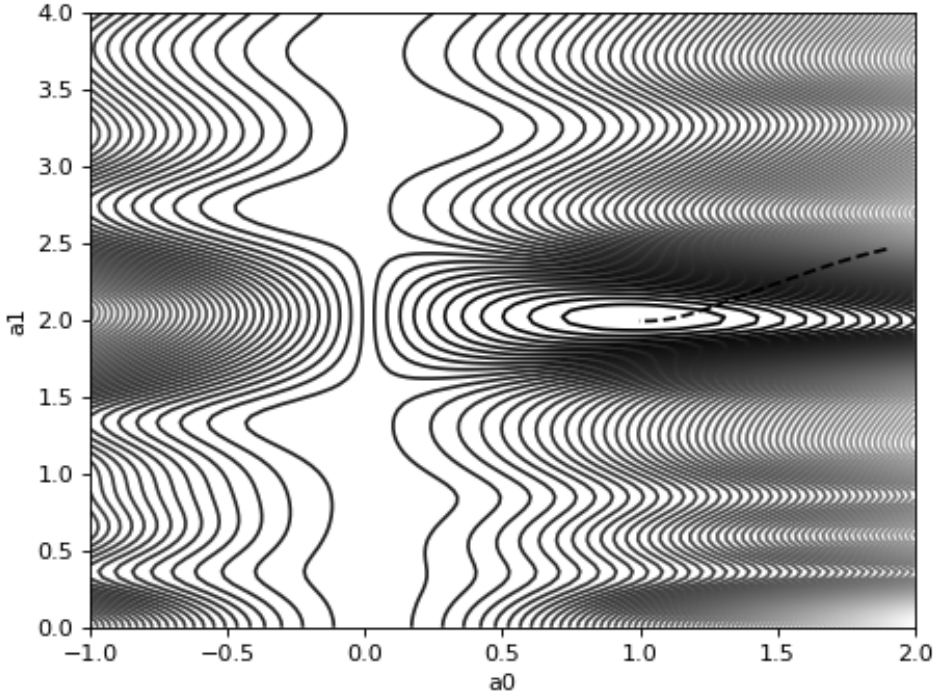
Pour suivre l'évolution des paramètres au cours des itérations, il est intéressant de représenter l'erreur en fonction des paramètres. La fonction suivante génère une carte de l'erreur :

```
def carte_erreur(f,x,y,sigma,a0,a1,delta_a0,delta_a1,M):
    N=len(x)
    a0_min=a0-delta_a0/2
    a1_min=a1-delta_a1/2
    da0 = delta_a0/M
    da1 = delta_a1/M
    E = numpy.zeros((M,M))
    for m in range(M):
        for n in range(M):
            e = 0
            for i in range(N):
                ecart = y[i]-f(x[i],a0_min+m*da0,a1_min+n*da1)
                e += ecart**2/sigma[i]**2
            E[n,m] = e
    list_a0 = numpy.linspace(a0_min,a0_min+delta_a0,M)
    list_a1 = numpy.linspace(a1_min,a1_min+delta_a1,M)

    return (list_a0,list_a1,E)
```

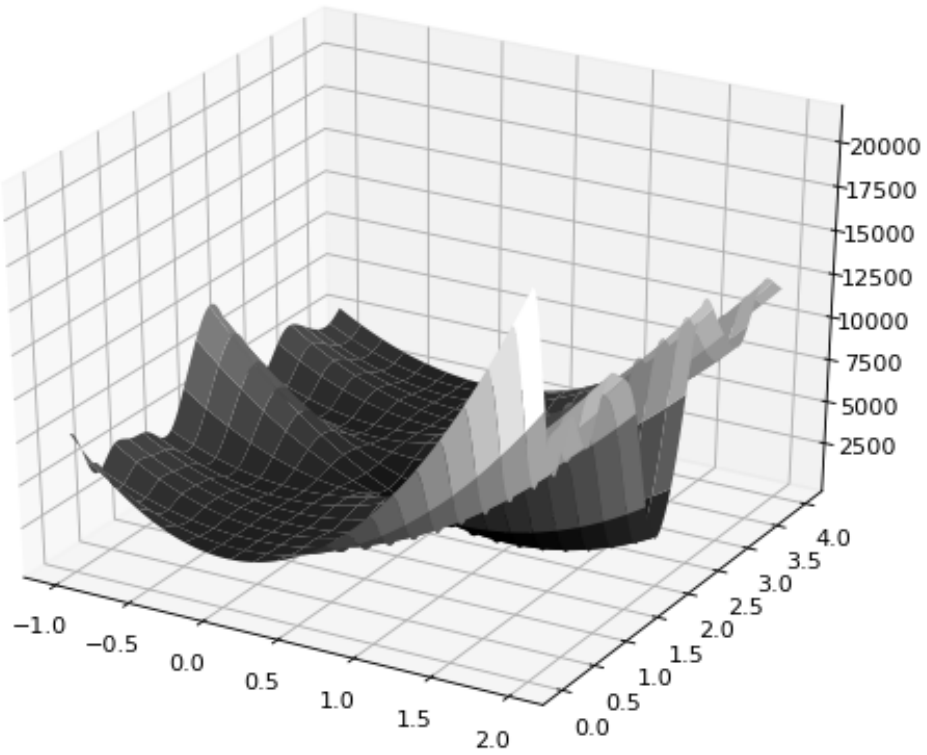
L'évolution des paramètres est suivie sur cette carte :

```
(list_a0,list_a1,E)=carte_erreur(f,x,y,sigma,0.5,2,3,4,100)
figure()
grid_a0,grid_a1=numpy.meshgrid(list_a0,list_a1)
levels = numpy.linspace(E.min(),E.max(),100)
contour(grid_a0,grid_a1,E,levels=levels,cmap=cm.gray)
plot(list_a00,list_a01,"k--")
xlabel("a0")
ylabel("a1")
```



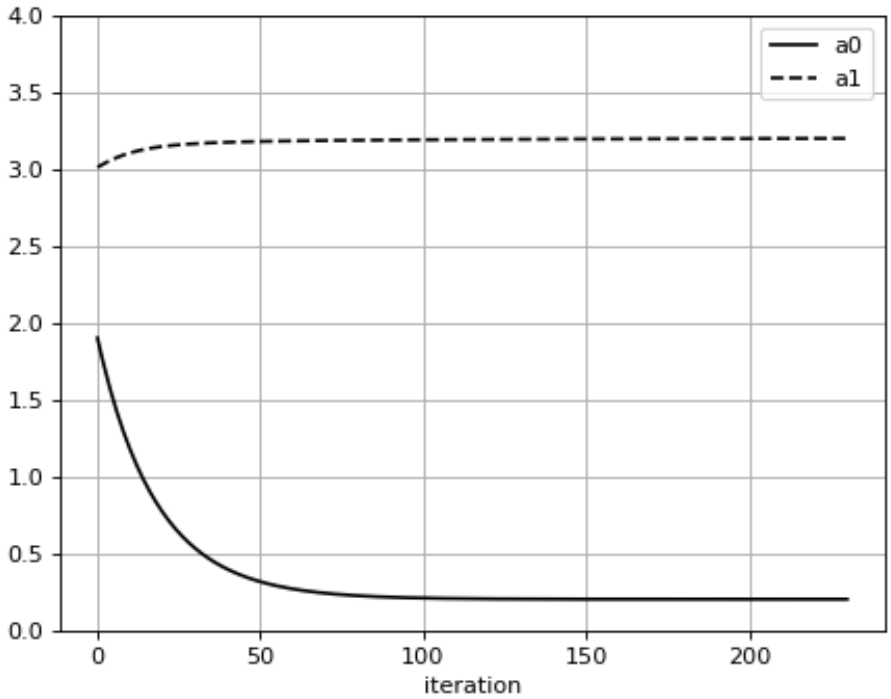
Voici aussi une représentation de l'erreur sous la forme d'une surface :

```
fig=figure()
ax = Axes3D(fig)
ax.plot_surface(grid_a0, grid_a1, E, rstride=5, cstride=5, cmap=cm.gray)
```

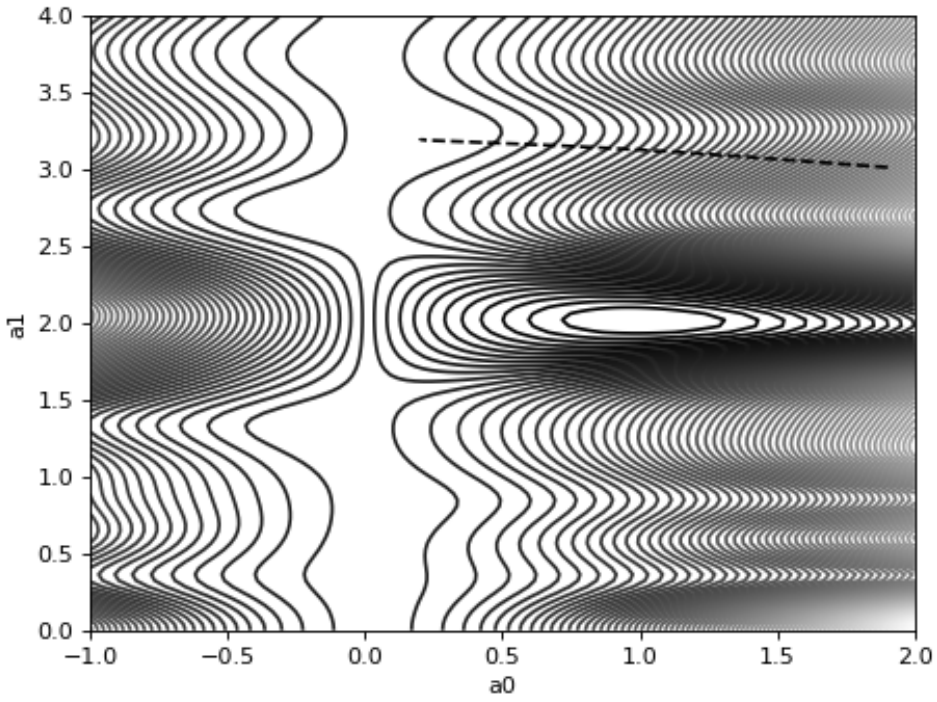


On remarque que la fonction erreur possède plusieurs vallées, avec un minimum absolu dans la vallée centrale. Sur cet essai, il y a bien convergence vers une valeur très proche des paramètres exacts car les valeurs initiales sont déjà dans la vallée correspondant au minimum absolu. Voyons un calcul avec des conditions initiales plus éloignées :

```
a0i=2
a1i=3.0
(list_a00,list_a01,list_e) = moindre_carre_non_lineaire_gradient(f,x,y,sigma,a0i,a1i,
figure()
plot(list_a00,"k-",label="a0")
plot(list_a01,"k--",label="a1")
xlabel("iteration")
legend(loc="upper right")
grid()
ylim(0,4)
```

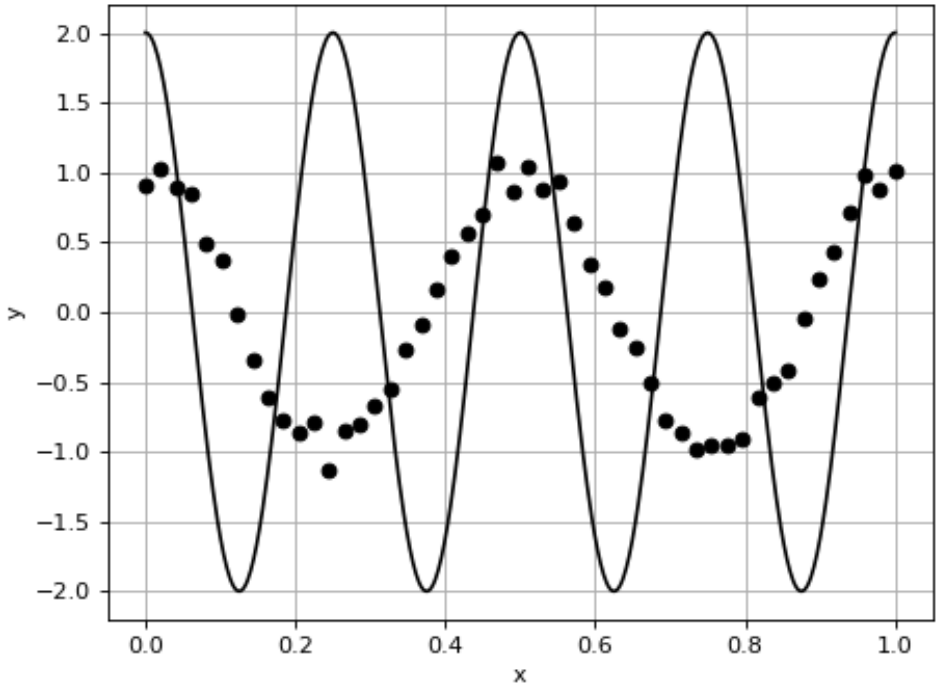



```
(list_a0,list_a1,E)=carte_erreur(f,x,y,sigma,0.5,2,3,4,100)
figure()
grid_a0,grid_a1=numpy.meshgrid(list_a0,list_a1)
levels = numpy.linspace(E.min(),E.max(),100)
contour(grid_a0,grid_a1,E,levels=levels,cmap=cm.gray)
plot(list_a00,list_a01,"k--")
xlabel("a0")
ylabel("a1")
```



Pour ses valeurs initiales des paramètres, la descente se fait vers un minimum local. Voyons à quelle fonction correspond ce minimum local :

```
figure()
list_x=numpy.linspace(0,1,500)
list_y=f(list_x,list_a0[-1],list_a1[-1])
plot(list_x,list_y,"k-")
plot(x,y,"ko")
grid()
xlabel("x")
ylabel("y")
```



Comme on le voit sur cet exemple, un minimum local peut être très loin de la solution optimale.