cours-optimisation-2

October 29, 2023

1 Résolution de problèmes d'optimisation II

Objectifs :

- Implémentation de la méthode du gradient conjugué pour une fonctionnelle quadratique
- Implémentation de la méthode de Lagrange-Newton pour un problème d'optimisation avec contraintes d'égalités

1.1 Méthode du gradient conjugué pour une fonctionnelle quadratique

Soit $A \in \mathbb{R}^{n\times n}$ une matrice symmétrique définie positive, et $b \in \mathbb{R}$. On pose :

$$f(x) = \frac{1}{2}x^T Ax - b^T x.$$

On considère le problème de minimisation :

$$f(\overline{x}) = \min_{x\in\mathbb{R}^n} f(x). \tag{1}$$

On rappelle que (1) admet une unique solution $\overline{x}$, qui de plus est l'unique solution de

$$A\overline{x} = b, \tag{2}$$

voir vos notes de cours. On remarque que

$$\nabla f(x) = Ax - b,$$

et donc (2) s'écrit simplement

$$\nabla f(x) = 0. \tag{3}$$

Il faut donc garder en tête que la méthode du gradient conjugué est à la fois une méthode pour résoudre un problème de minimisation particulier (1), et une méthode pour résoudre un système linéaire (2) avec une matrice symmétrique définie positive.

1.1.1 L'algorithme

La méthode est une méthode de descente de gradient à pas optimal. Plus particulièrement, on va déterminer une suite $(x_k)_{k\in\mathbb{N}}$ de points, une suite $(w_k)_{k\in\mathbb{N}}$ de directions de descentes et une suite $(\rho_k)_{k\in\mathbb{N}}$ de pas, telles que

$$x_{k+1} = x_k + \rho_{k+1} w_k. \tag{4}$$

Le pas ρ_{k+1} est celui qui minimise f au point x_k et dans la direction de descente w_k :

$$(5) \qquad f(x_k + \rho_{k+1} w_k) = \min_{\rho > 0} f(x_k + \rho w_k).$$

Une particularité de l'algorithme est que les directions de descente sont choisies pour être orthogonales à chaque itérations :

$$(6) \qquad w_k^T A w_{k'} = 0 \qquad \text{si } k \neq k'.$$

Notez que cette orthogonalité est pour le produit scalaire suivant : $(u, v) \mapsto u^T A v$ qui est adapté au problème.

On initialise l'algorithme en choisissant un point initial x_0.

Une fois x_k déterminé, on cherche à déterminer x_{k+1}. On pose u_k, le gradient de f en x_k:

$$(7) \qquad u_k = A x_k - b.$$

Si $u_k = 0$, alors x_k est la solution de (2) et on a fini. Sinon, on choisit la direction de descente suivante. Une première idée serait de prendre l'opposé du gradient : $-u_k$. Mais, on souhaite que la direction de descente soit orthogonale aux directions de descentes $w_0, ..., w_{k-1}$ des étapes précédentes. On choisit donc

$$w_k = -u_k + \sum_0^{k-1} \alpha_{k,i} w_i$$

avec les coefficients α_i choisis tels que $w_k A^T w_i = 0$ pour $i = 0, ..., k-1$. On affirme (ce que l'on démontrera plus bas pour ne pas alourdir l'explication) que

$$(8) \qquad \alpha_{k,i} = 0 \qquad \text{pour } i = 0, ..., k-2.$$

On a donc simplement

$$(9) \qquad w_k = -u_k + \alpha_k w_{k-1}$$

et l'on détermine α_k en utilisant que $w_k A w_{k-1} = 0$ afin que (6) soit satisfait, de sorte que

$$(10) \qquad \alpha_k = \frac{u_k^T w_{k-1}}{w_{k-1}^T A w_{k-1}}.$$

On cherche alors x_{k+1} sous la forme (4) avec ρ_{k+1} le pas optimal qui satisfait (5). On calcule par (7) que

$$f(x_k + \rho w_k) = \frac{1}{2}\rho^2 w_k^T A w_k + \rho(x_k^T A w_k - b^T w_k) + C_k, \qquad C_k = \frac{1}{2} x_k^T A x_k - b^T x_k.$$

Donc $\frac{d}{d\rho}(\rho \mapsto f(x_k + \rho w_k)) = \rho w_k^T A w_k + x_k^T A w_k - b^T w_k$. Comme A est définie positive, on a $w_k^T A w_k > 0$ et donc le minimum est atteint pour

$$\rho_{k+1} = \frac{b^T w_k - x_k^T A w_k}{w_k^T A w_k} = -\frac{w_k^T u_k}{w_k^T A w_k}$$

où l'on a utilisé (7) pour la seconde égalité. On peut montrer, cf cours, que l'algorithme converge en au plus n itérations vers la solution de (1)-(2).

1.1.2 Le code

Puisque si $\nabla f(x) = 0$ alors x est une solution de (1)-(2), on prendra $\|\nabla f(x_k)\|$ comme erreur dans l'algorithme. Celui-ci s'arrêtera donc dès que celle-ci sera plus petite qu'un seuil $\epsilon > 0$ défini au prealable.

La fonction `grad_conj(A,b,x0,K,e)` ci-dessous calcule une solution approchée de (1)-(2) par l'algorithme du gradient conjugué décrit précédemment.

[1]:
```
import numpy as np
def grad_conj(A,b,x0,K,e):
    k=0
    x=np.copy(x0) # l'utilisation de np.copy() permet de réaliser une copie␣
 ↪distincte de x0
    u=A@x-b   # u est egal au gradient de f au point xk
    w=-np.copy(u)    # w est wk direction de descente en xk. initialement pour␣
 ↪k=0 on a w=-nabla f(x0)=-u
    while k<K and np.linalg.norm(u)>e:
        rho=-np.sum(u*w)/(np.sum((A@w)*w))
        x=x+rho*w
        u=A@x-b
        alpha=np.sum(u*(A@w))/(np.sum((A@w)*w))
        w=u+alpha*w
        k=k+1
    if k<K:
        return(x)
    else:
        return("l'algorithme n'a pas converge")
```

On la teste ci-dessous sur l'exemple de la solution de

$$Ax = b, \qquad A = \begin{pmatrix} 10 & 1 & 3 & -1 \\ 1 & 10 & 1 & 1 \\ 3 & 1 & 10 & 1 \\ -1 & 1 & 1 & 10 \end{pmatrix}, \qquad b = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}.$$

[2]:
```
A1=np.array([[10,1,3,-1],[1,10,1,1],[3,1,10,1],[-1,1,1,10]])
b1=np.array([[1],[2],[3],[4]])
x0_1=np.array([[0],[0],[0],[0]])
x1=grad_conj(A1,b1,x0_1,10,0.001)
x1,A1@x1   # on verifie que A1@x1 est bien egal a [1,2,3,4]
```

[2]:
```
(array([[0.0532046 ],
        [0.13447043],
        [0.233736  ],
        [0.36845277]]),
 array([[0.99927167],
        [2.00009765],
```

```
[2.999897  ],
[3.99952956]]))
```

Remarques : (1) Un résultat théorique du cours assure que l'algorithme du gradient conjugué pour une fonctionnelle quadratique converge en au plus n étapes vers la solution exacte. Pourtant sur cet exemple après n=4 itérations l'algorithme n'a pas encore convergé. Cela est du à des erreurs numériques dans les opérations de multiplication, d'addition et de division effectuées dans l'algorithme.

(2) L'algorithme du gradient conjugué est une méthode très utile pour résoudre des systèmes linéaires, mais elle ne s'applique qu'aux matrices symmétriques définies positives.

1.1.3 Preuve de (8)

On démontre l'assertion (8) pour compléter l'explication de l'algorithme. On va d'abord montrer que

$$(11) \qquad u_k^T w_i = 0 \qquad \text{pour tous } i < k$$

en la montrant par récurrence sur $k \in \mathbb{N}$. L'assertion (11) est trivialement vraie pour $k = 0$. On suppose maintenant que (11) est vrai pour un $k \in \mathbb{N}$.

Alors, par définitions (4) et (5) de x_k et de ρ_{k+1}, on a que $\nabla f(x_{k+1})^T w_k = 0$, soit $u_{k+1}^T w_k = 0$. Or on a par (4) et (7) que

$$(12) \qquad u_{k+1} = Ax_{k+1} - b = A(x_k + \rho_k w_k) - b = u_k + \rho_k A w_k.$$

Pour $i \leq k-1$ on a que $u_k^T w_i$ par hypothèse de récurrence, et $w_k^T A w_i = 0$ par (6). Donc $u_{k+1}^T w_i = 0$. Donc (11) est vraie pour $k+1$.

Comme $u_k \in Vect(w_0, ..., w_k)$, on déduit de (11) que

$$(13) \qquad u_k^T w_i = 0 \qquad \text{pour tous } i < k.$$

Comme par (12) on a que $Aw_k = (u_{k+1} - u_k)/\rho_k$ et que $\rho_k \neq 0$ tant que l'algorithme ne s'est pas arrêté, on a par (13) que

$$u_k^T A w_i = 0 \qquad \text{pour tous } i \leq k-2.$$

Ceci implique les égalités désirées (8).

1.1.4 Exercice

Exercice 1.

On cherche à comparer les temps d'exécution des méthodes numériques de pivot de Gauss et de gradient conjugué pour des systèmes linéaires de grande taille. On considère ainsi pour $n \in \mathbb{N}$ le système linéaire

$$M_n x_n = b_n$$

avec $M_n \in \mathbb{R}^{n\times n}$ et $b_n \in \mathbb{R}^n$ donnés par

$$M_n = \begin{pmatrix} 2n & 1 & . & (1) \\ 1 & 2n & . & \\ . & . & . & . \\ (1) & . & 1 & 2n \end{pmatrix}, \qquad b_n = \begin{pmatrix} 1 \\ 2 \\ . \\ n \end{pmatrix}.$$

Comparer les temps d'exécution des fonctions `np.linalg.solve` et `grad_conj` pour des valeurs de n grandes. Que constatez-vous ?

1.2 Optimisation sous contrainte d'égalités : la méthode de Lagrange-Newton

Soit $f \in C^1(\mathbb{R}^n, \mathbb{R})$ et $g \in C^1(\mathbb{R}^n, \mathbb{R}^m)$ deux fonctions. On écrit $g(x) = (g_0(x), ..., g_{m-1}(x))$. Alors, en toute solution $\overline{x}$ du problème de minimisation sous contraintes d'égalités :

$$(2) \qquad \begin{cases} f(\overline{x}) = \min_{x\in\mathbb{R}^n,\ g(x)=(0,\dots,0)} f(x), \\ g(\overline{x}) = 0, \end{cases}$$

telle que $J(g)(\overline{x})$ est de rang m, il existe $(\overline{q_0}, ..., \overline{q_{m-1}}) \in \mathbb{R}^m$ tel que :

$$\nabla f(\overline{x}) + \sum_{i=0}^{m-1} \overline{q_i} \nabla g_i(\overline{x}) = \begin{pmatrix} 0 \\ ... \\ 0 \end{pmatrix}$$

(voir notes de cours sur multiplicateurs de Lagrange). L'égalité ci-dessus s'écrit $\nabla_x L(\overline{x}, \overline{q}) = 0$ pour $q = (q_0, ..., q_{m-1})$ et $L(x, q) = f(x) + \sum_{i=0}^{m-1} q_i g_i(x)$, et la fonction L s'appelle le Lagrangien de ce problème de minimisation sous contraintes d'égalités. Pour toute solution du problème de minimisation (2) pour laquelle $J(g)(\overline{x})$ est de rang m il existe donc $\overline{q} \in \mathbb{R}^m$ tel que $\overline{y} = (\overline{x}, \overline{q})$ est une solution du problème :

$$(3) \qquad F(\overline{y}) = 0$$

où

$$(4) \qquad \begin{array}{rccl} F: & \mathbb{R}^{n+m} & \to & \mathbb{R}^{n+m} \\ & y = (x, q) & \mapsto & \left(\nabla f(x) + \sum_{i=0}^{m-1} q_i \nabla g_i(x), g_0(x), ..., g_{m-1}(x)\right). \end{array}$$

La méthode de Lagrange-Newton pour résoudre (2) consiste à trouver $\overline{y} = (\overline{x}, \overline{q_0}, ..., \overline{q_{m-1}})$ une solution de (3) à l'aide d'un schéma de Newton-Raphson. Ensuite, puisque (3) peut avoir des solutions qui ne sont pas des solutions de (2) (par exemple des maxima de f sous la contrainte $g(x) = 0$), on vérifie que le point $\overline{x}$ trouvé de cette manière est bien une solution de (2).

Pour être sûr de ne pas faire d'erreurs dans le code, on procède par étapes. D'abord on code la fonction F. La fonction `F(Jf,g,n,m)` ci-dessous prend en entrée la fonction $x \mapsto \nabla f(x)$, la fonction g, la fonction $x \mapsto Jg(x)$, et les entiers $n, m \in \mathbb{N}$, et renvoie la fonction F donné par (4).

[162]:
```
def F(Jf,g,Jg,n,m):
    def F_aux(y):
        x=y[0:n]
        q=y[n:n+m]
        z=np.zeros([n+m])   # z est le vecteur retourne par F
        z[0:n]=Jf(x)
        for i in range(m):
```

```
            z[0:n]=z[0:n]+q[i]*Jg(x)[i]
        z[n:n+m]=g(x)
        return(z)
    return(F_aux)
```

On va tester notre algorithme sur l'exemple des fonctions $f(x) = x_0^2 - x_1^3 + x_0 x_1$ et $g(x) = x_0^2 + x_1^2 - 1$ (donc $n = 2$ et $m = 1$). On a alors $Jf(x) = \nabla f(x)^T = (2x_0 + x_1, -3x_1^2 + x_0)$ et $Jg(x) = (2x_0, 2x_1)$. Dans ce cas, on a :

$$F(x_0, x_1, q_0) = (2x_0 + x_1 + 2q_0 x_0, -3x_1^2 + x_0 + 2q_0 x_1, x_0^2 + x_1^2 - 1) \tag{5}$$

[163]:
```
def Jf1(x):
    return(np.array([2*x[0]+x[1],-3*((x[1])**2)+x[0]]))
def g1(x):
    return(np.array([(x[0])**2+(x[1])**2-1]))    # on met g sous forme␣
 ↪vectorielle pour que ce soit compatible avec le code de F
def Jg1(x):
    return(np.array([[2*x[0],2*x[1]]]))      # on met Jg sous forme vectorielle␣
 ↪pour que ce soit compatible avec le code de F et JF ci dessous
F1=F(Jf1,g1,Jg1,2,1)
F1(np.array([0,0,0])), F1(np.array([1,1,1])) # on vérifie sur des exemples que␣
 ↪F1 est bien la fonction ci-dessus
```

[163]:
```
(array([ 0.,  0., -1.]), array([5., 0., 1.]))
```

L'algorithme de Newton-Raphson nécessite de connaître JF, la matrice jacobienne de F. De (4) on déduit que :

$$JF(x,q) = \begin{pmatrix} \partial_{x_0x_0} f + \sum_{l=0}^{m-1} q_l \partial_{x_0x_0} g_l & \dots & \partial_{x_0x_{n-1}} f + \sum_{l=0}^{m-1} q_l \partial_{x_0x_{n-1}} g_l & \partial_{x_0} g_0 & \dots & \partial_{x_0} g_{m-1} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \partial_{x_0x_{n-1}} f + \sum_{l=0}^{m-1} q_l \partial_{x_0x_{n-1}} g_l & \dots & \partial_{x_{n-1}x_{n-1}} f + \sum_{l=0}^{m-1} q_l \partial_{x_{n-1}x_{n-1}} g_l & \partial_{x_{n-1}} g_0 & \dots & \partial_{x_{n-1}} g_{m-1} \\ \partial_{x_0} g_0 & \dots & \partial_{x_{n-1}} g_0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \partial_{x_0} g_{m-1} & \dots & \partial_{x_{n-1}} g_{m-1} & 0 & \dots & 0 \end{pmatrix}.$$

On constate que ceci peut s'écrire sous la forme :

$$JF(x,q) = \begin{pmatrix} M_1 & M_3 \\ M_2 & (0) \end{pmatrix} \tag{6}$$

avec $M_1 = Hf + \sum_{i=0}^{m-1} q_i Hg_i$, $M_2 = Jg$ et $M_3 = (Jg)^T$. La fonction `JF(Hf,Jg,Hg,n,m)` ci-dessous prend en entrée la fonction $x \mapsto Hf(x)$, la fonction $x \mapsto Jg(x)$, la fonction $x \mapsto Hg(x)$, et les entiers $n, m \in \mathbb{N}$, et renvoie la fonction JF donné par (6).

[164]:
```
def JF(Hf,Jg,Hg,n,m):
    def JF_aux(y):
        x=y[0:n]
        q=y[n:n+m]
```

```
        M=np.zeros([n+m,n+m])  # M est la matrice retournee par JF
        M[0:n,0:n]=Hf(x)
        for i in range(m):
            M[0:n,0:n]=M[0:n,0:n]+q[i]*Hg(x)[i]
        M[n:n+m,0:n]=Jg(x)
        M[0:n,n:n+m]=np.transpose(Jg(x))
        return(M)
    return(JF_aux)
```

Pour l'exemple précédent donné par (5) un calcul direct donne

$$JF(x_0, x_1, q_0) = \begin{pmatrix} 2+2q_0 & 1 & 2x_0 \\ 1 & -6x_1+2q_0 & 2x_1 \\ 2x_0 & 2x_1 & 0 \end{pmatrix}$$

et on vérifie ci-dessous que c'est bien le résultat renvoyé par `JF(Hf,Jg,Hg,n,m)`.

[165]:
```
def Hf1(x):
    return(np.array([[2,1],[1,-6*x[1]]]))
def Hg1(x):
    return(np.array([[[2,0],[0,2]]]))
JF1=JF(Hf1,Jg1,Hg1,2,1)
JF1(np.array([0,0,0])),JF1(np.array([1,1,1]))  # on vérifie sur des exemples␣
 ↪que JF1 est bien la fonction ci-dessus
```

[165]:
```
(array([[2., 1., 0.],
        [1., 0., 0.],
        [0., 0., 0.]]),
 array([[ 4.,  1.,  2.],
        [ 1., -4.,  2.],
        [ 2.,  2.,  0.]]))
```

[166]:
```
Newton_Raphson(F1,JF1,np.array([-1,-1,-1]),0.0001,10)
```

[166]:
```
array([-0.66215608, -0.74936829, -1.56585468])
```

On reprend ci-dessous le code pour l'algorithme de Newton-Raphson vu lors du cours sur les systèmes non linéaires.

[167]:
```
def Newton_Raphson(Func,JFunc,y0,eps,N):
    e=2*eps
    n=0
    y=y0
    while e>eps and n<N:
        y=y-np.linalg.solve(JFunc(y),Func(y))
        e=np.linalg.norm(Func(y))
        n=n+1
    if n<N:
        return(y)
```

```
    else:
        return("l'alogrithme n'a pas converge")
```

Et on utilise les fonctions `F(Jf,g,Jg,n,m)`, `JF(Hf,Jg,Hg,n,m)` et `Newton_Raphson(Func,JFunc,x0,eps,N)` ci-dessus pour écrire une fonction `Lag_New(Jf,Hf,g,Jg,Hg,x0,q0,eps,N)` ci-dessous qui implémente l'algorithme de Lagrange-Newton et résout le problème (2) en initialisant à $y_0 = (x_0, q_0)$.

[168]:
```
def Lag_New(Jf,Hf,g,Jg,Hg,x0,q0,eps,N):
    n=len(x0)
    m=len(q0)
    y0=np.zeros(n+m)
    y0[0:n]=x0
    y0[n:n+m]=q0
    Func_aux=F(Jf,g,Jg,n,m)
    JFunc_aux=JF(Hf,Jg,Hg,n,m)
    return(Newton_Raphson(Func_aux,JFunc_aux,y0,eps,N))
```

On le teste sur l'exemple précédent pour déterminer la solution du problème de minimisation :

$$\begin{cases} \overline{x_0}^2 - \overline{x_1}^3 + \overline{x_0 x_1} = \min_{x\in\mathbb{R}^2,\ \|x\|^2=1} x_0^2 - x_1^3 + x_0 x_1, \\ \overline{x_0}^2 + \overline{x_1}^2 = 1 \end{cases}$$

et l'on constate que l'algorithme converge effectivement, mais que plusieurs limites sont possibles.

[169]:
```
Lag_New(Jf1,Hf1,g1,Jg1,Hg1,np.array([1,1]),np.array([1]),0.
↪000001,10),Lag_New(Jf1,Hf1,g1,Jg1,Hg1,np.array([-1,-1]),np.array([-1]),0.
↪000001,10)
```

[169]:
```
(array([ 0.95462344,  0.29781562, -1.15598596]),
 array([-0.66215112, -0.74937033, -1.5658605 ]))
```

Exercice 2.

1. Représenter graphiquement la fonction $f(x_0, x_1) = x_0^2 - x_1^3 + x_0 x_1$ pour $ -2 x_0,x_1 2$ à l'aide de la fonction `pcolormesh` de pyplot. Déterminer grossièrement ou se situe le minimiseur de f sous la contrainte $\|x\|^2 = 1$ (ce qui signifie que x est sur le cercle unité).

2. Déterminer une valeur approchée de ce minimiseur à l'aide de la fonction `Lag_New(Jf,Hf,g,Jg,Hg,x0,q0,eps,N)`.

[ ]: