tp1-2223

November 22, 2022

[ ]:
```
from IPython.display import display, Latex
from IPython.core.display import HTML
%reset -f
%matplotlib inline
%autosave 300
from math import *
from matplotlib.pylab import *
from time import time
import numpy as np
```

1 Un exemple en dimension finie

On applique les différentes méthodes d'optimisation numérique vues en cours à un problème quadratique simple.

On veut résoudre numériquement le problème modèle suivant:

$$(P) \qquad \inf\left\{J(x) := \frac{1}{2}\langle A.x, x\rangle - \langle b, x\rangle \;:\; x\in\mathbb{R}^3\right\}$$

où

$$A := \begin{pmatrix} 1 & 4 & 3 \\ -3 & 6 & 3 \\ -1 & 0 & 7 \end{pmatrix} \qquad et \qquad b := \begin{pmatrix} -1 \\ 1 \\ -1 \end{pmatrix}.$$

1.0.1 Petite étude mathématique

Montrer que J est strictement convexe. Ecrire les conditions d'optimalité et calculer la solution optimale théorique de (P).

La condition d'optimalité est

$\frac{1}{2}(A + A^T)x - b = 0$ et on trouve :

[ ]:
```
A=np.array([[1,4,3],[-3,6,3],[-1,0,7]])
S = (A+np.transpose(A))/2. # S = A symétrisée

b=np.transpose(np.array([[-1,1,-1]]))
sol=np.linalg.solve(S,b)
print("\nla solution est :\n",sol)
```

Dans la suite, on pourra prendre comme test la distance à l'unique solution optimale de (P).

1.0.2 La méthode de la plus grande pente à pas constant pour ce problème.

On prend pour point initial $x^0 = 0$ et pour précision $\varepsilon = 10^{-2}$ (pour tester), puis $\varepsilon = 10^{-6}$.

On testera diverses valeurs pour le pas choisi et on comparera la vitesse de convergence (c'est-à-dire le nombre d'itérations nécessaires avant l'arrêt de l'algorithme).

Une première version simple de la methode de la plus grande pente est :

[ ]:
```
# pas / step
h=2e-1

# precision
eps=1e-2

# point initial
x0 = np.transpose(np.array([[0,0,0]]))

# main iteration
t=time()
x=x0
cpt=0
dist=1.
while dist > eps :
#while np.linalg.norm( S @ x -b ):
#while np.linalg(x-sol) > eps:
    y=x-h*(S @ x -b)
    dist=np.linalg.norm(x-y)
    x=y
    cpt += 1

print("\n","solution calculée :\n",x)
print(f"\n en {cpt} itérations pour un pas h={h}\n")
print(f"pour un temps de calcul égal à {time()-t:.4f}s")
```

Une version un peu meilleure (car plus claire) :

[ ]:
```
def gradJ(x):
    return S @ x - b

# pas / step
h=2e-1

# precision
eps=1e-2

# point initial
```

```
x0 = np.transpose(np.array([[0,0,0]]))

x=x0
g=gradJ(x)
t=time()
cpt=0
dist=1.
while dist > eps :
#while np.linalg.norm(g) > eps :
    y=x-h*g
    dist=np.linalg.norm(x-y)
    x=y
    g=gradJ(x)
    cpt += 1


print("\n","solution calculée :\n",x)
print(f"\n en {cpt} itérations pour un pas h={h}\n")
print(f"pour un temps de calcul égal à {time()-t:.4f}s")

# solution exacte :
sol=np.linalg.solve(S,b)
print("solution exacte :\n",sol)

# ecart:
print("\n","l'erreur est || x - sol ||=",np.linalg.norm(x-sol))
```

1.0.3 Méthode à pas optimal

Programmer la méthode de la plus grande pente à pas optimal pour ce problème. Pour cela, on rappelle que la solution du problème

$$\min \{j(t) = J(x - t\nabla J(x)) \,:\, t > 0\}$$

est

$$\rho_{opt} = \frac{\langle \nabla J(x), \nabla J(x) \rangle}{\langle A\nabla J(x), \nabla J(x) \rangle}.$$

[ ]:

```
```

1.0.4 Méthode avec recherche linéaire

Programmer la méthode de la plus grande pente en utilisant la recherche linéaire avec critère d'Armijo pour calculer le pas à chaque étape.

Faire plusieurs essais selon les valeurs des paramètres α et c_1 de cette recherche linéaire et comparer les vitesses de convergence (nombre d'itérations pour chaque recherche linéaire, et pour l'algorithme dans son ensemble).

[ ]:

1.0.5 5) Méthode proximale

Programmer la méthode prox : on rappelle que cela consiste, à l'étape n, à résoudre de manière approchée le problème

$$\inf\left\{J(x) + \frac{1}{2\,\delta_n}\|x - x^n\|^2 \,:\, x \in \mathbb{R}^3\right\}$$

et à choisir pour x^{n+1} la solution approchée trouvée.

Pour résoudre le problème d'optimisation apparaissant à chaque itération on choisira une méthode à pas constant ou avec recherche linéaire.

[ ]:

1.0.6 6) Méthode de quasi-Newton

Programmer la méthode de quasi-Newton BFGS pour ce problème, en employant une recherche linéaire avec le critère d'Armijo.

1.0.7 7) Calcul du gradient par différence fini

Reprendre les questions précédentes en employant la formule suivante pour dériver de manière approchée J :

$$\frac{\partial J}{\partial x_i}(x) \simeq \frac{J(x+\delta e_i) - J(x-\delta e_i)}{2\delta}$$

où δ est une constante petite.

Cela revient à employer le code suivant pour le gradient :

[ ]:

2 Un exemple en dimension infinie

Dans ce qui suit $\Omega =]-1,1[$. On définit la fonctionnelle J sur $H^1(\Omega)$ par

$$v \mapsto J(v) := \frac{1}{2}\int_{-1}^{1} (1 + 10\,(1-x^2))\,(v'(x))^2 dx$$

On étudie alors le problème

$$(P) \qquad \inf\left\{J(v) \,:\, v \in H^1(\Omega),\ v = \overline{u} \text{ sur } \partial\Omega\right\}.$$

On prendra pour la suite $\overline{u} : x \mapsto x$.

2.0.1 Résolution théorique

Montrer que J est convexe, faiblement sci, et que le problème (P) a au moins une solution. Est-elle unique ?

Ecrire la formulation faible pour ce problème.

2.0.2 Résolution numérique

Pour résoudre ce problème de manière approchée, on le discrétise par la méthode des éléments finis. Pour $N \geq 1$ fixé, on considère l'ensemble $\mathcal{A}_N$ des fonctions affines par morceaux $f : [-1, 1] \to \mathbb{R}$ telles que $f(-1) = -1$, $f(1) = 1$ et

$$\forall n \in \{0, \ldots, N-1\}, \qquad f \text{ affine sur } \left[-1 + \frac{2n}{N}, -1 + \frac{2(n+1)}{N}\right].$$

Toute fonction $f \in \mathcal{A}_N$ est alors caractérisée par le vecteur $y \in \mathbb{R}^{N+1}$ donné par

$$\forall n \in \{0, \ldots, N\}, \qquad y_n = f\left(-1 + \frac{2n}{N}\right).$$

L'ensemble des vecteurs ainsi obtenus est alors

$$\mathcal{B}_N = \{y \in \mathbb{R}^{N+1} \,:\, y_0 = -1,\, y_N = 1\}.$$

On doit alors résoudre le problème :

$$(P_N) \qquad \inf\{G(y) \,:\, y \in \mathcal{B}_N\}$$

où $G(y) = J(f)$ pour la fonction affine $f \in \mathcal{A}_N$ associée au vecteur y.

Pour commencer, on va donc déclarer les variables du problème discrétisé sous Python, et afficher les valeurs pour l'utilisateur. On définit donc N, y_A et y_B :

[ ]:
```python
N=3;
#print("nombre de points de discretisation : N =",N,"\n");
yA=-1;
yB=1;
#print("valeurs au bord : y_A =",yA,"et y_B =",yB);
```

On aura aussi besoin d'un vecteur initial pour la méthode de la plus grande pente, on prend l'interpolation affine entre les valeurs y_A et y_B sur les $N+1$ points de discrétisaton de $[-1, 1]$. Ainsi le vecteur y est de taille $N+1$, avec $y_1 = y_A$ et $y_{N+1} = y_B$ fixés, et les autres points sont obtenus par interpolation :

[ ]:
```python
def init(N,yA,yB):
    y=[((N-i)*yA+i*yB)/N for i in range(N+1)]
    y=np.array(y,float)
    return y
```

```
y=init(N,yA,yB)
#print("y=",y)

a=-1
b=1
x=linspace(a,b,N+1)
#print("\n"+"x=",x)

#plot(x,y);
```

On transforme y en une matrice colonne :

[ ]:
```
y=np.array(y,ndmin=2)
y=np.transpose(y)
#print(y)
```

Remarque : pour faire l'opération inverse, on peut par exemple écrire :

[ ]:
```
# methode 1
z=[y[i][0] for i in range(len(y))]
print(z)
# methode 2
z=np.transpose(y)
z=z[0]
print(z)
```

Reprendre les questions 2) à 6) précédentes pour ce nouveau problème (P), en prenant plusieurs valeurs pour le paramètre de discrétisation N (on considérera des petites valeurs $N = 3$, $N = 9$, ou grandes $N = 50$…) et en utilisant le gradient approché de la question 7).

Pour le calcul de l'intégrale qui apparait dans J, on peut utiliser la méthode de Simpson sur chaque intervalle de la discrétisation.

[ ]:
```
```