© *J.-B.A.K. <jean-baptiste.apoung@math.u-psud.fr>*

Note 1 (Changement dans le déroulement)**.**

Pour la séance 2, on devra procéder comme suit :
- Achever la fiche TP1
- Puis entamer la présente fiche (TP2) avec pour objectif de l'achever avant la fin de la séance.

TP 2 : Optimisation sous contrainte : un problème d'obstacle.

On souhaite résoudre numérique un problème d'optimisation donné sous la forme :

$$\text{Minimiser } J(\mathbf{u}) \text{ sur } K = \{\mathbf{v} \in \mathbb{R}^N : \varphi_1(\mathbf{v}) \leq 0, \ldots, \varphi_d(\mathbf{v}) \leq 0\}. \tag{1}$$

Dans cette fiche nous allons entre autres :
- nous familiariser à une bibliothèque `Python` de résolution directe de ce type de problème
- reformuler ce problème afin afin de tirer partie des méthodes de minimisation sans contrainte.

Partie - 1 *Le problème modèle et sa discrétisation par éléments finis*

Soit f et g deux fonctions continues données sur $[0,1]$. On souhaite résoudre le problème d'obstacle suivant :

$$\begin{cases} \min\limits_{u \in K} J(u) \\ J(u) = \displaystyle\int_0^1 \left(\frac{1}{2}u'(x)^2 - f(x)u(x)\right) dx \\ K = \left\{u \in H^1(]0,1[) : u(0) = a, u(1) = b,\ u(x) \geq g(x)\ \forall x \in [0,1]\right\} \end{cases} \tag{2}$$

Comme dans le TP1, on discrétise ce problème par éléments finis. On considère alors un maillage de $[0,1]$ donné par $0 = x_0 < x_1 < \ldots < x_i < \ldots < x_{N+1} = 1$. On pose $h_i = x_{i+1} - x_i, i = 0, \ldots, N+1$ et $h = \max\limits_{0 \leq i \leq N+1} h_i$.
Dès lors on pose

$$K_h = \Big\{v \in C^0([0,1]) : v(0) = a, v(1) = b,\ v_{|_{[x_i,x_{i+1}]}}(x) = v(x_i) + (x - x_i)\frac{v(x_{i+1}) - v(x_i)}{h_i}, i = 0, \ldots, N, \\ v(x_i) \geq g(x_i),\ i = 0, \ldots, N+1\Big\} \tag{3}$$

On approche le problème (2) par le suivant :

$$\begin{cases} \min\limits_{u_h \in K_h} J(u_h) \\ J(v_h) = \displaystyle\int_0^1 \left(\frac{1}{2}v_h'(x)^2 - f(x)v_h(x)\right) dx \end{cases} \tag{4}$$

Q-1 : [**Explicitation du problème discret**]

Q-1-1 : Expliciter ce problème (4) en utilisant notamment la formule des trapèzes pour calculer les intégrales et montrer qu'il peut se mettre sous la forme suivante

$$\begin{cases} \min\limits_{u_N \in K_N} J_N(u_N) \\ K_N = \left\{v \in R^N : v_i \geq g(x_i) : \forall\ 1 \leq i \leq N\right\} \end{cases} \tag{5}$$

où J_N est à expliciter.(*On pourra se servir du TP1* .)

Q-1-2 : Calculer le gradient ∇J_N de J_N.

Q-1-3 : En s'inspirant du cours montrer que le problème (5) admet une unique solution.

Q-2 : **[Résolution directe de (5).]**

Q-2-1 : Implémenter les fonctionnelles J_N respectivement ∇J_N, à travers les deux fonctions PYTHON suivantes :

```
def J(U, a, b, x, fx)
```

```
def DJ(U, a, b, x, fx)
```

Q-2-2 : Pour $a = 0, b = 0$, $N = 100$, $f(x) = 1, g(x) = \max\left(1.5 - 20(x - 0.6)^2, 0\right)$

— résoudre le problème (5) à l'aide de **`scipi.optimize.minimize`**
On pourra compléter le bout de script suivant :

```
import numpy as np
from scipy.optimize import minimize
...
''' Definir ou importer f, g, J  et DJ  '''
...
N = 100
a, b, x  = 0, 0, np.linspace(0,1, N + 2)

fx, gx  = f(x), g(x)

Jf = lambda u : J(u, a, b, x, fx)
DJf = lambda u : DJ(u, a, b, x, fx)

cons = ({'type': 'ineq',
         'fun' : lambda x: x - gx[1:-1],
         'jac' : lambda x: np.eye(np.size(x))})

u = np.zeros(n)
res = minimize(Jf, u, method='SLSQP', jac=DJf, constraints=cons, tol=1e-8,
               options={'xtol': 1e-8, 'disp': True,'maxiter':5000})

# res.x  contient la solution
```

— Et représenter sur le même graphique la solution ainsi que l'obstacle. Tester aussi avec $f(x) = \pi^2 \sin(\pi x)$.

Q-3 : **[Validation]**

Q-3-1 : *Vérifier* que la solution de (2) pour $a = b = 0$ vérifie

$$\left.\begin{array}{rcl} -u''(x) & \geq & f(x), \\ u(x) & \geq & g(x), \\ (-u''(x) - f(x))(u(x) - g(x)) & = & 0, \end{array}\right\} \quad \text{sur }]0,1[, \text{ et } u(0) = 0,\ u(1) = 0. \tag{6}$$

La première équation traduit une concavité maximale de la fonction u. La deuxième équation représente l'obstacle : on veut que la solution u soit au dessus de g. La troisième équation traduit le fait que l'on a au moins égalité dans une des deux équations précédentes : soit on résout $-u''(x) = f(x)$, soit $u(x) = g(x)$, et u est sur l'obstacle.

Q-3-2 : Faire varier N et vérifier si la solution numérique obtenue à la question **Q-2** *approche* (i.e. vérifie les propriétés attendues) la solution de l'équation (6).

Partie - 2 *Méthode de pénalisation : prise en main*

La méthode de pénalisation consiste à remplacer le problème de minimisation sous contrainte (2) par une suite de problèmes de minimisation sans contrainte qui converge vers (2).
Soit $\eta > 0$ donné, et J_N^η défini par :

$$J_N^\eta(\mathbf{u}) = J_N(\mathbf{u}) + \frac{1}{\eta}\sum_{i=1}^{N}(\max(g_i - u_i, 0))^2.$$

Le minimiseur $\mathbf{u}_N^\eta$ de J_N^η sur $\mathbb{R}^N$ converge vers $\mathbf{u}_N$ quand η tend vers 0.
On admet que le gradient de J_N^η est donné par :

$$\nabla J_N^\eta(\mathbf{u}) = \nabla J_N(\mathbf{u}) - \left(\frac{2}{\eta}\text{signe}(g_i - u_i)\max(g_i - u_i, 0)\right)_{i=1,\ldots,N}$$

où la fonction signe est définie par : $\text{signe}(x) = \begin{cases} -1 & \text{si} \quad x < 0 \\ 0 & \text{si} \quad x = 0 \\ 1 & \text{si} \quad x > 0 \end{cases}$

Q-4 : **Mise en oeuvre.** En adaptant la fonction **`gradient_fixe.py`** du TP1, implémenter cet algorithme à travers une fonction une fonction **`penalisation.py`**

Code Listing 1 – Fichier penalisation.py

```
def penalisation(J, DJ, gN, eta, u0, rho, epsilon, iterMax, store):
"""
ENTREES
   J       : fonctionnelle à minimiser.
   DJ      : le gradient de la fonctionnelle a minimiser.
   gN      : vecteur identifiant le convex K i.e  K={v : v[i] >= gN[i],i=1...d}
   eta     : parametre de penalisation destine a tender vers 0
   u0      : valeur initiale.
   rho     : pas fixe.
   epsilon : test d'arret.
   iterMax : nombre maximal d'iterations autorisees.
   store   : parametre controllant le type de stockage dans u. Il prend les valeurs 0 ou 1.
  SORTIES
    u      : dernier terme de la suite des iteres uk si store = 0 ou tous les termes si store = 1.
    iter   : nombre d'iterations effectuees.
"""
```

Q-5 : **Validations.** Tester cette fonction pour les valeurs numériques de la question **Q-**3, en prenant : $N = 50$ et $\eta = 10^5, 10^4, 10^3, 10^2, 10, 1, 0.1$. Afficher à l'aide de la fonction **`print`** le nombre d'itérations ainsi que le temps de calcul pour chaque η. Tracer sur une même figure les iso-solutions approchées $\mathbf{u}_N^\eta$, ainsi que le graphe de la fonction g et le graphe de la solution de $-u''(x) = 1$. Que constatez-vous pour η grand ? Pour η petit ?

Note 2 (Indications).

On rappelle que si l'on dispose des deux fonctions

```
def penalite(u,gN,etha)
```

```
def grad_penalite(u,gN,eta)
```

dont le premier implémente le terme de pénalisation et le second son gradient, alors le listing suivant implémente la méthode du gradient pour le problème pénalisé

```
Jf = lambda U : J(U) + penalite(U, gN, eta) ;
DJf = lambda U :DJ(U) + grad_penalite(U, gN, eta) ;
u, iter = gradient_fixe(Jf,DJf,u0,rho,epsilon,iterMax,store);
```

Il faut remarquer que dans le cas du problème considéré, la pénalisation apporte un terme quadratique ! Et par conséquent une possibilité d'utiliser un pas pas optimal.

Q-6 : **Conclusion.** Déduire alors la nécessité d'une méthode variant le paramètre de pénalisation η.