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

Présentation

On se propose dans la série des Tps qui vont suivre de résoudre numériquement un problème du type :

$$\min_{u\in K} J(u) \tag{1}$$

et K un espace fonctionnel.

Dans toute la suite, nous considérons, sans nuire à la généralité $\Omega =]0,1[$ et nous placerons dans les cas suivants :

— TP1 :

$$\begin{cases} J(u) = \displaystyle\int_0^1 \left(\frac{1}{2}u'(x)^2 - f(x)u(x)\right) dx \\ K = \left\{u \in H^1(]0,1[) : u(0), u(1) \text{ donnés}\right\} \end{cases} \tag{2}$$

— TP2 :

$$\begin{cases} J(u) = \displaystyle\int_0^1 \left(\frac{1}{2}u'(x)^2 - f(x)u(x)\right) dx \\ K = \left\{u \in H^1(]0,1[) : u(0), u(1) \text{ donnés}, \ u(x) \geq \psi(x)\ \forall x \in [0,1]\right\} \end{cases} \tag{3}$$

— TP3, TP4 :

$$\begin{cases} J(u) = \displaystyle\int_0^1 \left(\frac{1}{2}u'(x)^2 - f(x)u(x)\right) dx \\ K = \left\{u \in H^1(]0,1[) : u(0), u(1) \text{ donnés}\ , \ u''(x) \geq 0\ \forall x \in]0,1[\right\} \end{cases} \tag{4}$$

Dans chacun des cas nous nous focaliserons sur les schémas numériques de résolution des problèmes prenant en compte le cadre fonctionnel du problème continu et faisant usage des méthodes d'optimisation numérique.

— Dans le TP1, nous regarderons les méthodes de **gradient (à pas fixe - optimal, variables)** et la méthode du **gradient conjugué**
— Dans le TP2, nous regarderons la méthode du **gradient projeté**, et introduirons la **méthode de pénalisation**
— Dans le TP3, nous approfondirons la méthode de pénalisation, évoquerons la méthode **d'Uzawa**
— Dans le TP4, nous aborderons la méthode d' **Active-Set** .

Note 1.

— Attention la fonctionnelle $J(\cdot)$ est susceptible de changer dans les énoncés.
— La mise en oeuvre se fera avec le langage `Python` (pour lequel vous êtes beaucoup plus familiers).

Note 2.

Un cadre fonctionnel adapté pour approcher la solution de ces problème est offert par la méthode des éléments finis de Lagrange :

— On se donne une subdivision de $0 = x_0 < x_1 < \ldots < x_i < \ldots < x_{N+1} = 1$ de $[0,1]$.
— On définit l'espace éléments finis P_k Lagrange :

$$\mathcal{L}_h^k = \{v_h \in c^0([0,1]) : v_h|_{[x_i,x_{i+1}]} \text{ est polynômiale de degré } k\ \forall\, i = 0,\ldots,N\}$$

Ces espaces sont de bonnes approximations de dimensions finies de $H^1(]0,1[)$ (voir Cours).
— On approche alors
  — u par $u_h = \sum_{j=0}^{N_k} u_i\varphi_i$ où $\mathcal{L}_h^k = vect\{\varphi_0,\ldots,\varphi_{N_k}\}$
  — $J(v)\,\forall v \in H^1(]0,1[)$ par $J_h(v_h)\,\forall v_h \in \mathcal{L}_h^k$.
  — K par $K_h^k =$ " $K \cap \mathcal{L}_h^k$ " (*contraintes généralement prises ponctuellement*).
— On espère voir u_h converger vers u lorsque h tend vers 0.

TP1 : Optimisation sans contrainte en dimension finie. Equation de Laplace.

Partie - 1 *Résolution de l'équation de Laplace*

On se place ici dans le cas où $K = \{v \in H^1(]0,1[) : v(0) = a, v(1) = b\}$.

Le maillage est donné par $0 = x_0 < x_1 < \ldots < x_i < \ldots < x_{N+1} = 1$. Et on pose $h_i = x_{i+1} - x_i, i = 0, \ldots, N+1$ et $h = \max_{0 \leq i \leq N+1} h_i$.

On utilise les éléments finis $P_1 Lagrange$ c'est-à-dire $k = 1$.

Dès lors on pose $K_h = \{v \in C^0([0,1]) : v(0) = a, v(1) = b, v(x) = v(x_i) + (x - x_i)\frac{v(x_{i+1}) - v(x_i)}{h_i}\ \forall x \in [x_i, x_{i+1}], i = 0, \ldots, N\}$

On approche le problème (1) par le suivant : $\min_{u_h \in K_h} J(u_h)$

Q-1 : Expliciter ce problème et montrer qu'il est équivalent, si l'on utilise une formule de trapèzes pour le calcul des intégrales, au suivant

$$\begin{cases} \min\limits_{u_N \in R^N} J_N(u_N) \\ J_N(V) = \sum\limits_{i=0}^{N} \left(\dfrac{(V_{i+1} - V_i)^2}{2h_i} - \dfrac{h_i}{2}\left(f_{i+1}V_{i+1} + f_i V_i\right) \right) \end{cases} \tag{5}$$

où on a posé $f_i = f(x_i), i = 0, \ldots, N+1, h_i = x_{i+1} - x_i, i = 0, \ldots N$, et V_{N+1} resp. V_0 lorsqu'il est rencontré est remplacé par b resp. a.

Q-2 : Montrer que ce problème (5) peut se mettre sous la forme :

$$A_N \mathbf{u}_N = \mathbf{f}_N \tag{6}$$

où A_N et $\mathbf{f}_N$ sont à préciser.

Q-3 : Y-a-t'il une relation entre la solution du problème (5) et celle du problème (7) ci-dessous discrétisé par éléments finis P_1 Lagrange sur le même maillage avec la même formule de quadrature ?

$$\begin{cases} -u''(x) = f(x), \quad x \in]0,1[, \\ u(0) = a \quad \text{et} \quad u(1) = b. \end{cases} \tag{7}$$

Q-4 : Validation de l'approximation

On considère $f(x) = -2, a = 0, b = 1.$ de sorte que la solution exacte de (2) est $u(x) = x^2$.

Résoudre le système linéaire (6) en utilisant **`scipy.linalg.solve`**, pour différentes valeurs de N et vérifier que la solution de (5) converge bien pour la norme de $H^1(]0,1[)$ vers la solution u de (2) lorsque h tend vers 0.
(*On pourra prendre pour chaque N un maillage uniforme de pas $h = \frac{1}{N+1}$*)
(*On peut aussi directement résoudre (5) en utilisant* ***`scipy.optimize.minimize`***)

Q-5 : Vers la résolution directe de (5)

Dans toute la suite du TP on prendra $f = 1, a = 0, b = 0$.

Implémenter les fonctionnelles J_N respectivement ∇J_N, à travers les deux fonctions PYTHON suivantes :

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

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

On rappelle que pour **`a,b,x`** et **`fx`** donnés, l'instruction suivante de PYTHON ,
`JN = lambda V : J(V, a, b, x, fx)`, définit une fonction de la seule variable V.

Partie - 2 *Méthode du gradient à pas fixe*

On rappelle l'algorithme du gradient à pas fixe pour une fonctionnelle $J : \mathbb{R}^N \to \mathbb{R}$, un point de départ $\mathbf{u}^0$, un pas ρ et un test d'arrêt ε préalablement définis :

Méthode du gradient à pas fixe
Initialiser le résidu r^0 à 1 et le compteur k à 0.
Tant que le résidu est plus grand que ε et que le compteur n'est pas trop grand :
- calculer la descente $\mathbf{w}^k = -\nabla J(\mathbf{u}^k)$,
- poser $\mathbf{u}^{k+1} = \mathbf{u}^k + \rho \mathbf{w}^k$,
- calculer le résidu $r^{k+1} = ||\mathbf{u}^{k+1} - \mathbf{u}^k||$,
- incrémenter le compteur.

Q-6 : **Mise en oeuvre.** Implémenter cet algorithme à travers une fonction de prototype :

Code Listing 1 – Fichier gradient_fixe.py

```
def gradient_fixe(J, DJ, u0, rho, epsilon, iterMax, store):
"""
ENTREES
  J       : fonctionnelle a minimiser.
  DJ      : le gradient de la fonctionnelle a minimiser.
  u0      : valeur initiale.
  rho     : pas fixe.
  epsilon : test d'arret.
  iterMax : nombre maximal d'iterations autorisees.
  store   : parametre pilotant 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.
"""
```

Les questions qui suivent permettront de valider l'implémentation et de comprendre certaines propriétés de la méthode. On créera un script **`scriptTP1_fixe.py`** pour répondre à ces questions.

Q-7 : **Validations : cas $N = 2$.**

Q-7-1 : Tracer sur une même figure les courbes de niveaux de J_2 ainsi que le champ de vecteurs ∇J_2 sur le pavé $[-10, 10]\times[-10, 10]$. On utilisera les fonctions **`matplotlib.pyplot.contour`** et **`matplotlib.pyplot.quiver`**.

Q-7-2 : Calculer les itérations $\mathbf{u}^k = (u_1^k, u_2^k)$ données par l'algorithme de gradient à pas fixe, et tracer sur la même figure que précédemment la ligne qui relie les $\mathbf{u}^k$. On prendra $\mathbf{u}^0 = (8, 4)$, $\rho = 0.1$ et $\varepsilon = 10^{-12}$.

Q-8 : **Validations : cas N quelconque.**

Q-8-1 : Pour $N = 2, 5, 20, 50$. Afficher à l'aide de la fonction **`fprintf`** le nombre d'itérations ainsi que le temps de calcul pour chaque N. Tracer sur une même figure les solutions approchées $\mathbf{u}_N$, ainsi que la solution exacte de (7). On prendra $\rho = 0.1$, $\varepsilon = 10^{-12}$.

Q-8-2 : Reprendre l'expérience précédente pour $\rho = 0.5$, puis $\rho = 1$. Que constate-t-on ? Peut-on choisir le pas ρ arbitrairement ?

Partie - 3 *Méthode du gradient à pas optimal*

On rappelle l'algorithme du gradient à pas optimal pour une fonctionnelle $J : \mathbb{R}^N \to \mathbb{R}$, un point de départ $\mathbf{u}^0$ et un test d'arrêt ε préalablement définis :

Méthode du gradient à pas optimal
Initialiser le résidu r^0 à 1 et le compteur k à 0.
Tant que le résidu est plus grand que ε et que le compteur n'est pas trop grand :
- calculer la descente $\mathbf{w}^k = -\nabla J(\mathbf{u}^k)$,
- calculer $\rho^k \geq 0$ qui minimise $\rho \mapsto J(\mathbf{u}^k + \rho \mathbf{w}^k)$,
- poser $\mathbf{u}^{k+1} = \mathbf{u}^k + \rho^k \mathbf{w}^k$,
- calculer le résidu $r^{k+1} = ||\mathbf{u}^{k+1} - \mathbf{u}^k||$,
- incrémenter le compteur.

Q-9 : Mise en oeuvre.

Q-9-1 : Minimisation mono-dimensionnelle associée.

Pour déterminer le pas optimal ρ_k fournir trois approches comme décrites dans le Listing 2 :

Code Listing 2 – Détermination du pas optimal

```
def optimal_alpha_analytique(J, DJ, uk, wk, rho0):
    #Par un calcul explicite tenant compte du caratere quadratique de J
    #_____________________________________________________________________
def optimal_alpha_newton(J, DJ, uk, wk, rho0) :
    # Par la émthode de Newton
    #_____________________________________________________________________
def optimal_alpha_section_doree(J, DJ, uk, wk, rho0):
     # Par la émthode de la section édore vue en cours (voir Annexe)
     #_____________________________________________________________________
     # Chacune de ces fonctions prend en arguments:
     # J la fonctionnnelle a minimiser, DJ son gradient, uk et wk les solution
     # et direction de descente courrantes et enfin rho0 le pas de édpart.
     # Elles retournent chacune rho: le pas optimal comme édcrit dans l'algorithme
```

Q-9-2 : Fournir une fonction

Code Listing 3 – Fichier gradient_optimal.py

```
def gradient_optimal(J, DJ, u0, rho, epsilon, iterMax, store, optim1D):
"""
  ENTREES
   J       : fonctionnelle a minimiser.
   DJ      : le gradient de la fonctionnelle a minimiser.
   u0      : valeur initiale.
   rho     : valeur initiale du pas (n'est pas forcement utilise).
   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.
   optim1D : entier (e votre convenance) identifiant le type de methode pour determiner le pas optimal.
  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.
    rhoL   : liste des pas optimaux generes par les iterations.
 """
    return u, iter, rhoL
```

On créera un script **`scriptTP1_optimal.py`** pour la validation.

Q-10 : Validations : cas $N = 2$. Reprendre les expériences effectuées dans la méthode du gradient à pas fixe.

Q-11 : Validations : cas N quelconque.

Q-11-1 : Reprendre la question **Q-3-1**.

Q-11-2 : Modifier la fonction **`gradient_optimal.py`** de sorte qu'elle retourne aussi les directions de descente w^k que l'on rangera dans les colonnes d'une matrice W.

Q-11-3 : Pour le pas optimal obtenu de manière analytique, et pour $N = 30$,

a) Comparer les pas obtenus avec la valeur optimale (donnée en cours) du pas dans la méthode du gradient à pas fixe (lorsque J est quadratique).

b) Calculer $W' * W$ et conclure (W' est la transposée de W).

Partie - 4 *Méthode du gradient conjugué*

Cette méthode n'est valable que pour des fonctionnelles de la forme $J(\mathbf{u}) = \frac{1}{2}(A\mathbf{u}, \mathbf{u}) - (\mathbf{b}, \mathbf{u})$, où A est une matrice symétrique définie positive. L'algorithme du gradient conjugué pour une telle fonctionnelle, avec un point de départ $\mathbf{u}^0$ et un test d'arrêt ε préalablement définis, est donné par :

Méthode du gradient conjugué
Initialiser le résidu r^0 à $||A\mathbf{u}^0 - \mathbf{b}||$, la descente $\mathbf{w}^0$ à $-(A\mathbf{u}^0 - \mathbf{b})$, et le compteur k à 0.
Tant que le résidu est plus grand que ε et que le compteur n'est pas trop grand :

- calculer $\rho^k = -\dfrac{(A\mathbf{u}^k - \mathbf{b}, \mathbf{w}^k)}{(A\mathbf{w}^k, \mathbf{w}^k)}$,
- poser $\mathbf{u}^{k+1} = \mathbf{u}^k + \rho^k \mathbf{w}^k$,
- calculer la nouvelle descente $\mathbf{w}^{k+1} = -(A\mathbf{u}^{k+1} - \mathbf{b}) + \dfrac{||A\mathbf{u}^{k+1} - \mathbf{b}||^2}{||A\mathbf{u}^k - \mathbf{b}||^2}\mathbf{w}^k$,
- calculer le résidu $r^{k+1} = ||A\mathbf{u}^{k+1} - \mathbf{b}||$,
- incrémenter le compteur.

Q-12 : Réécrire l'algorithme de sorte à ne faire intervenir que J et ∇J mais pas A ni b.
Montrer qu'un test d'arrêt $\|\nabla J(u_k)\| \leq \epsilon \|\nabla J(0)\|$ est beaucoup plus approprié (0 est le vecteur nul de même taille que u_k).

Q-13 : Mise en oeuvre.
Implémenter cet algorithme à travers une fonction de prototype :

Code Listing 4 – Fichier gradient_conjugue.py

```
def gradient_conjugue(J, DJ, u0, rho, epsilon, iterMax, store):
"""
ENTREES
  J       : fonctionnelle e minimiser.
  DJ      : le gradient de la fonctionnelle a minimiser.
  u0      : valeur initiale.
  rho     : valeur initiale du pas (n'est pas forcement utilise).
  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.
   rhoL   : liste des pas generes par les iterations.
"""
    ...
    return u, iter, rhoL
```

On créera un script **`scriptTP1_conjugue.py`** pour la validation.

Q-14 : Validations : cas $N = 2$. Reprendre les expériences effectuées dans la méthode du gradient à pas fixe.

Q-15 : Validations : cas N quelconque.

Q-15-1 : Reprendre la question **Q-3-1**.

Q-15-2 : Modifier la fonction **`gradient_conjugue.py`** de sorte à retourner aussi les directions de descente w^k et les gradients (ou vecteurs résidus) $DJ(u^k)$ que l'on rangera par colonnes respectivement dans les matrices W et Z.

Q-15-3 : Pour $N = 30$,

a) Comparer les pas obtenus avec ceux obtenus par la méthode du gradient à pas optimal.
b) Calculer $Z' * Z$, $Z' * W$, $Z' * (A * W)$, $W' * (A * W)$ (W' est la transposée de W. On rappelle que la hessienne $\nabla^2 J$ est une constante que nous désignons par A). Justifier alors pourquoi la méthode du gradient conjugué convergera beaucoup plus vite que celle du gradient à pas optimal.

Utilitaires pour l'optimisation sans contrainte en dimension 1.

Méthode de la section dorée

Cette méthode est valable uniquement pour une fonction :
- *à valeurs réelles,*
- *dont on connaît un intervalle $[a, b]$ sur lequel elle admet un unique minimum local.*

Le principe de la méthode est un peu semblable à celui de la dichotomie, mais à chaque étape on calcule la valeur de la fonction en deux points de l'intervalle $[a, b]$ de départ, définis par :

$$a' = a + \frac{b-a}{\tau^2} \quad \text{et} \quad b' = a + \frac{b-a}{\tau} \quad \text{avec} \quad \tau = \frac{1+\sqrt{5}}{2}.$$

Algorithme de la section dorée pour minimiser une fonction f
Initialiser le compteur k à 0, l'erreur `err` à $b - a$.
Tant que `err` $> \varepsilon$ (tolérance choisie) :
- *calculer a' et b',*
- *évaluer $f(a')$ et $f(b')$,*
- *si $f(a') > f(b')$: poser $a = a'$*
  si $f(a') < f(b')$: poser $b = b'$
  si $f(a') = f(b')$: poser $a = a'$ et $b = b'$,
- *calculer la nouvelle erreur `err` $= b - a$,*
- *incrémenter le compteur.*

Code Listing 5 – Fichier methode_section_doree_recursive.py

```
 def methode_section_doree_recursive(a, b, f, epsilon):
     tau = (1+sqrt(5))/2
     k =  0
     err =  b - a
     delta = 1e-36
     bp = a + (b-a)/tau
     ap = a+ b - bp
     fap = f(ap)
     fbp = f(bp)
     fa = f(a)
     fb = f(b)
     iter = 0
     return util_methode_section_doree_recursive(a,ap,bp,b, fa,fap,fbp, fb,f,epsilon,iter)
# ---------------------- utilitaire -------------------------

def util_methode_section_doree_recursive(a, ap, bp, b,fa,fap,fbp,fb, f,epsilon,ki):
    if( (b-a) < (epsilon*(abs(ap) + abs(bp) ))) :
        return (b+a)/2 , ki
   else :
       if( fap  >  fbp):
           ki = ki+1
           return util_methode_section_doree_recursive(ap,bp,ap + b -bp,b, fap,fbp,f(ap+b-bp), fb, f,epsilon↩
                ,ki)
       else :
           ki = ki + 1
           return util_methode_section_doree_recursive(a,a+bp-ap,ap,bp, fa,f(a+bp-ap),fap, fbp,f,epsilon,ki)
```

Méthode de Newton en dimension 1

Cette méthode est valable pour des fonctions dont on connaît une approximation des zéros.

L'algorithme de Newton-Raphson permet de trouver un point en lequel une fonction f s'annule, connaissant une approximation x^0 de ce point :

Algorithme de Newton-Raphson en dimension 1
Initialiser le compteur `it` *à* 0, *l'erreur* `err` *à* 1.
Tant que `it` < `itmax` *et que* `err` $> \varepsilon$ *(tolérance choisie) :*

- *calculer le prochain candidat pour le zéro de* f *:* $x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)}$,
- *calculer l'erreur* `err` $= |f(x^k)|$,
- *incrémenter le compteur.*

Code Listing 6 – Fichier methode_newton.py

```
 def methode_newton(f, fp, x0, epsilon, itermax) :
"""
 ENTREES:
  f  : fonction donc on cherche une le zero
  fp : la éédrive de f
  x0 : solution initiale
  epsilon : étolrance
  itermax : nombe maximal d'éitrations éautorises
 SORTIES:
    x : solution la valeur éapproche du zero de f
    iter: le numbre d'éitrations éeffectues
"""
      iter =  0
      err  = 1.
      x = x0
      fx = f(x)
      while ( (iter < itermax) and (err > epsilon)) :
          fpx = fp(x)
          delta = fx / fpx
          x  = x - delta
          fx = f(x)
          err  = min(abs(delta)/(abs(x)+1.), abs(fx));
          iter = iter + 1
      return x, iter
```