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

TP4 : Optimisation sous contrainte en dimension finie : reconstruction d'un signal par projection sur les fonctions convexes.

Note 1.

- Ceci n'est pas une solution de la fiche de TP.
- Ce sont simplement des scripts pour alléger dans la réalisation du projet.
  Remarquer que les contraintes sont mises à l'échelle. En Tps, il a été proposé des implémentations optimales de certaines de ces fonctions.
- Les scripts sont ceux demandés en TPs et sont fournis ici sans commentaires.
  Les commentaires donnés en Tps suffisent pour les comprendre, sinon solliciter moi par mail.

Table des matières

1 Construction du problème discret

Code Listing 1 – Fonctions permettant de construire l'énergie

```
 def JL(u,x, ubruit):
    v = np.array([0]+u.tolist()+[0])
    vb = np.array([0]+ubruit.tolist()+[0])
    h = x[1:] - x[0:-1]
    v = v - vb
    w = v[0:-1]**2 + v[0:-1] * v[1:] + v[1:]**2
    r = np.dot(h/3.0,w)
    return r

def DJL(u,x, ubruit):
    n = len(u)
    v = np.array([0]+u.tolist()+[0])
    vb = np.array([0]+ubruit.tolist()+[0])
    h = x[1:] - x[0:-1]
    v = v - vb
    r = np.zeros(n)
    for i in range(1,n+1):
        r[i-1] = (h[i-1] * v[i-1] + 2*(h[i-1] + h[i]) * v[i] + h[i] * v[i+1])/3
    return r


def JH(u, x, ubruit):
    v = np.array([0]+u.tolist()+[0])
    vb = np.array([0]+ubruit.tolist()+[0])
    v = v - vb
    h = x[1:] - x[0:-1]
    w = (v[1:] - v[0:-1])**2
    r = np.dot(1./h,w)
    return r

def DJH(u, x, ubruit):
    n = len(u)
    v = np.array([0]+u.tolist()+[0])
    vb = np.array([0]+ubruit.tolist()+[0])
    v = v - vb
    h = x[1:] - x[0:-1]
    r = np.zeros(n)
    for i in range(1,n+1):
        ri = (v[i]- v[i-1])/h[i-1] + (v[i] - v[i+1])/h[i]
        r[i-1] = 2.0 * ri
    return r
```

Code Listing 2 – Fonctions nécessaires pour les contraintes

```
def Contraintes(u, x, bruit):
    n = len(u)
    v = np.array([0]+u.tolist()+[0])
    h = x[1:] - x[0:-1]
    r = np.zeros(n)
    for i in range(1,n+1):
        r[i-1] = -v[i-1]/h[i-1] + (1/h[i-1] + 1/h[i]) * v[i]  - v[i+1]/h[i]
    return r

def GradContraintes(u, x, bruit):
""" Les contraintes sont lineaire """
    n = len(u)
    a = np.eye(n)
    for i in range(n):
        a[:,i] = Contraintes(a[:,i],x,bruit)
    return a
```

Code Listing 3 – Fonctions nécessaires pour la pénalisation

```
def penalite(u, contraintes, gradContraintes, eta):
    phi = contraintes(u)
    v = np.max([phi, 0*u],0)
    return np.dot(v,v) / eta

def gradPenalite(u, contraintes, gradContraintes, eta):
    phi = contraintes(u)
    v = np.max([phi, 0*u],0)
    gradT = np.transpose(gradContraintes(u))
    return gradT.dot(v)*2/eta
```

Code Listing 4 – Fonctions utilitaires

```
def extractMatrixFromQuadForm(DJ, n):
""" Extraction de la matrice d'une forme quadratique"""
    a = np.eye(n)
    b = DJ(np.zeros(n))
    for i in range(n):
        a[:,i] = DJ(a[:,i]) - b
    return a


def matrixSpectrum(A):
""" Valeurs propres exxtreme d'une matrice sysm. def. pos."""
    lam = np.linalg.eig(A)[0]
    return np.min(lam), np.max(lam)
```

2 Résolution directe

Code Listing 5 – Probleme modele

```
def problemeModele(signal, N, amplitude = 0.5):
    """
    Genere l'interpole P1 sur un maillage de N+2 points de [0,1]
    d'un signal bruite issu d'un signal d'entree
    ENREE :
       signal    : une fonction definie sur [0,1] et a valeur reelle
       N         : nombre de noeuds internes du maillage
       amplitude : amplitude du bruit a introduire
    SORTIE:
       ub        : un signal bruite du signal d'entree sur un maillage de [0,1]
       x         :les noeuds du maillage considere de [0,1]
    """
    x = np.linspace(0,1, N+2)
    bruit =  (-1.0 + 2 * random.rand(N+2)) /2
    ub = signal(x) + amplitude * bruit
    ub[0]  = 0
    ub[-1] = 0
    return ub, x
```

Code Listing 6 – Résolution directe

```
def testReconst():
    N = 100
    signal = lambda x: -np.sin(np.pi*x) #x*(x-1)
    bruit , x = problemeModele(signal,N,1.000)
    Jf = lambda u : (JL(u, x, bruit[1:-1]) + JH(u, x, bruit[1:-1]))
    DJf = lambda u : (DJL(u, x, bruit[1:-1]) +  DJH(u, x, bruit[1:-1]))
    cons = ({'type': 'ineq',
             'fun' : lambda u: -Contraintes(u,x,bruit[1:-1]),
             'jac' : lambda u: -GradContraintes(u,x,bruit[1:-1])})
    u = np.zeros(N)
    res = minimize(Jf, u, method='SLSQP', jac=DJf, constraints=cons, tol=1e-12,
                   options={'xtol': 1e-8, 'disp': True,'maxiter':5000})
    plt.plot(x, signal(x), '--g', x, bruit,'-y', x, [0] + res.x.tolist()+[0],'r-.')
    plt.xlabel('$x$');
    plt.ylabel('signal(x)')
    plt.title("Reconstruction d'un signal par projection H1 sur les fonctions convexes");
    plt.legend(['signal','bruite','reconstruit'])

    plt.savefig('solution1.png',format='png')
    plt.figure(2)
    plt.plot(np.arange(N),Contraintes(res.x,x,bruit[1:-1]),'*', np.arange(N), 0 * u)
    plt.show()
    print("erreur {0:11.9e}".format(np.linalg.norm(bruit[1:-1] - res.x)))
```

3 Résolution par la méthode d'uzawa

Rappel de l'algorithme :

Pour un point de départ $\mathbf{u}^0$, un choix initial de $\lambda^0 = (\lambda_1^0, \ldots, \lambda_d^0)$, une tolérance ε, et un pas ρ donné, cet algorithme s'écrit de la façon suivante :

Méthode d'Uzawa
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 $\mathbf{u}^{k+1}$ le minimiseur de $\mathbf{v} \mapsto L_{\lambda^k}(\mathbf{v})$,
- poser $\lambda^{k+1} = (\max(\lambda_i^k + \rho\varphi_i(\mathbf{u}^{k+1}), 0))_{i=1\ldots d}$,
- calculer le résidu $r^{k+1} = ||\mathbf{u}^{k+1} - \mathbf{u}^k||$,
- incrémenter le compteur.

Pour mettre en oeuvre cet algorithme,

1. il faut à chaque itération résoudre le problème de minimisation sans contrainte associé.
   Ici on deux choix,
   - utiliser le module d'optimisation de python,
   - tirer partie du fait que $L_{\lambda^k}(\mathbf{v})$ est quadratique. En effet, on remarque qu'on peut écrire

   $$L_{\lambda^k}(V) = \frac{1}{2}(V - Ub)^T Q(V - Ub) + (\lambda^k)^T AV$$

   On peut alors construire les matrices A et Q et la solution du problème est obtenue par résolution du système linéaire

   $$Q(V - Ub) + A^T\lambda^k = 0$$

   On peut aussi dans un souci d'optimisation du coût calculs, envisager de stocker Q sous forme factorisée LU ou Cholesky par exemple.
2. Il faut aussi sélectionner le pas dans une plage acceptable : $\rho \in]0, \rho_{max}[$
   Ici, en revenant à la définition de $L_\lambda(V) = J(V) + \lambda^T\varphi(V)$ et en désignant par :
   - α la constante d'ellipticité de la J :

   $$< \nabla J(V_2) - \nabla J(V_1), V_2 - V_1 > \geq \alpha \|V_2 - V_1\|^2$$

   - M la constante de Lipschitz de φ :

   $$\|\varphi(V_2) - \varphi(V_1)\| \leq M \|V_2 - V_1\|$$

   Alors, la méthode d'Uzawa convergera pour tout pas $\boxed{0 < \rho < \frac{2\alpha}{M^2}}$

Note 2.

Dans le cas particulier du problème sous la main, on peut prendre pour
- α la plus petite valeur propre de Q
- M la plus grande valeur propre de A

Code Listing 7 – Uzawa

```
def Uzawa(J, DJ, contraintes, Gradcontraintes, u0, lamb0, rho, tol, iterMax):
    res = 1.
    iters = 0
    u = 1 * u0
    lamb = 1 * lamb0
    conv = False

    # ICI ON EXTRAIT LES MATRICES POUR RESOLUTION  EXACTE DU PROBLEME SANS CONTRAINTE
```

```
    Q = extractMatrixFromQuadForm(DJ, len(u))
    A = extractMatrixFromQuadForm(contraintes, len(u))
    b = DJ(0*u)

    while conv == False:

        """STEP1:  RESOLUTION DU PROBLEME SANS CONTRAINTE """

        # APPROCHE 1

        #Jf = lambda u : J(u) + lamb.dot(contraintes(u))
        #DJf = lambda u : DJ(u) + lamb.dot(Gradcontraintes(u))
        #res = minimize(Jf, u, method='SLSQP', jac=DJf, tol=1e-12,
        #          options={'disp': True,'maxiter':500})
        #u = res.x

        # APPROCHE 2:
        u = np.linalg.solve(Q, - lamb.dot(A) -b ) # Grace a python, on s'epargne  np.transpose(a).dot(lamb)

        """ STEP 2:  MISE A JOUR """
        lamb = lamb + rho * contraintes(u)
        lamb = np.max([lamb, 0*lamb],0)

        """ STEP 3: TEST DE CONVERGENCE """
        iters += 1
        res = np.linalg.norm(u - u0)
        u0 = 1 * u
        conv = (res < tol or iters >= iterMax)
        if(not(conv)):
            print("iteration uzawa : {0:d} residu : {1:10.3e}".format(iters, res))
    return u, iters
```

Code Listing 8 – Plage de pas admissible pour Uzawa

```
def plageUzawaRho(J, DJ, contraintes, Gradcontraintes, N):
    u = np.zeros((N,1))
    A = Gradcontraintes(u)              # Les contraintes sont supposees affine
    Q = extractMatrixFromQuadForm(DJ,N) # On suppose que J est quadratique
    lminA, lmaxA = matrixSpectrum(A)
    lminQ, lmaxQ = matrixSpectrum(Q)
    return 2 * lminQ/(lmaxA * lmaxA)
```

Code Listing 9 – Test Uzawa

```
def testReconstUzawa():
    N = 100
    rho = 1.0
    tol = 1e-6
    iterMax = 10000

    signal = lambda x: -np.sin(np.pi*x) #x*(x-1)
    bruit , x = problemeModele(signal,N,1.000)

    Jf = lambda u : (JL(u, x, bruit[1:-1]) + 0 * JH(u, x, bruit[1:-1]))
    DJf = lambda u : (DJL(u, x, bruit[1:-1]) + 0 * DJH(u, x, bruit[1:-1]))
    Cont = lambda u: Contraintes(u,x,bruit[1:-1])
    GradCont = lambda u: GradContraintes(u,x,bruit[1:-1])

    """  Preparation pour initialisation de l'algorithme uzawa. On peut faire mieux ici """

    u0 = np.zeros(N)
    lamb0 = np.zeros(N)

    rho = plageUzawaRho(Jf, DJf, Cont, GradCont, N)/2  # on prend un pas fixe dans la plage autorisee

    #print (rho)

    u, it = Uzawa(Jf, DJf, Cont, GradCont, u0, lamb0, rho, tol, iterMax)

    plt.plot(x, signal(x), '--g', x, bruit,'-y', x, [0] + u.tolist()+[0],'r-.')
    plt.xlabel('$x$');
    plt.ylabel('signal(x)')
    plt.title("Reconstruction d'un signal par projection H1 sur les fonctions convexes");
    plt.legend(['signal','bruite','reconstruit'])
```

4 Résolution par la méthode de pénalisation

Ici on fournit simplement une mise en oeuvre de la pénalisation demandée dans le TP2.

Attention ceci correspond à une seule itération de la méthode de pénalisation demandée dans le présent TP.

Car en effet, on s'attend dans le présent TP, plutôt à un algorithme dont le squelette aurait la forme du Listing 11 :

Code Listing 10 – Pénalisation demandée au TP2

```
def penalisation(J, DJ, contraintes, Gradcontraintes,
                        u0, rho, eta, tol, iterMax, store=0):

    Jf = lambda u : J(u) + penalite(u,contraintes,Gradcontraintes,eta)

    DJf = lambda u : DJ(u) + gradPenalite(u,contraintes,Gradcontraintes,eta)

    u = 1 * u0

    #u, iters =  gradient_optimal(Jf, DJf, u0, rho, tol, iterMax, store = 0)


    res = minimize(Jf, u, method='SLSQP', jac = DJf, tol = 1e-9,
                   options={'disp': True,'maxiter':500})
    u = res.x
    iters = res.nit

    #Approximation du Multiplicateur de lagrange

    lamb =  2 * np.max([contraintes(u),0*u],0)/eta

    return u, lamb, iters
```

Il faut modifier la fonction ci-dessus pour résoudre le problème du présent TP, de manière identique à la méthode d'UZAWA.

Code Listing 11 – Pénalisation

```
def Penalisation(J, DJ, contraintes, Gradcontraintes,  \
                 u0, eta0, rho, tol, iterMax):

    # STEP0 : INITIALISATION

    res = 1.
    iters = 0
    u = 1 * u0
    eta = 1 * eta0
    conv = False

    while conv == False:

        # STEP1 : RESOLUTION DU PROBLEME SANS CONTRAINTE
        # --> u

        # STEP2 : MISE A JOUR DU MULTIPLICATEUR DE LAGRANGE
        # -->  lamb =  2 * np.max([contraintes(u),0*u],0)/eta


        # STEP 3 : MISE A JOUR DU PARAMETRE DE PENALISATION
        # -- > eta

        # STEP4 : TEST DE CONVERGENCE

    return ....
```

Le test ce fait comme pour la méthode d'uzawa.