td1a_correction_session9

July 1, 2022

1 1A.algo - Optimisation sous contrainte (correction)

Un peu plus de détails dans cet article : Damped Arrow-Hurwicz algorithm for sphere packing.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

On rappelle le problème d'optimisation à résoudre :
$\begin{cases} \min_U J(U) = u_1^2 + u_2^2 - u_1 u_2 + u_2 \\ \textit{sous contrainte } \theta(U) = u_1 + 2u_2 - 1 = 0 \textit{ et } u_1 \geqslant 0.5 \end{cases}$
Les implémentations de l'algorithme Arrow-Hurwicz proposées ici ne sont pas génériques. Il n'est pas suggéré de les réutiliser à moins d'utiliser pleinement le calcul matriciel de numpy.

1.1 Exercice 1 : optimisation avec cvxopt

Le module cvxopt utilise une fonction qui retourne la valeur de la fonction à optimiser, sa dérivée, sa dérivée seconde.

$$\begin{array}{rcl} f(x,y) & = & x^2 + y^2 - xy + y \\ \frac{\partial f(x,y)}{\partial x} & = & 2x - y \\ \frac{\partial f(x,y)}{\partial y} & = & 2y - x + 1 \\ \frac{\partial^2 f(x,y)}{\partial x^2} & = & 2 \\ \frac{\partial^2 f(x,y)}{\partial y^2} & = & 2 \\ \frac{\partial^2 f(x,y)}{\partial x \partial y} & = & -1 \end{array}$$

Le paramètre le plus complexe est la fonction `F` pour lequel il faut lire la documentation de la fonction solvers.cp qui détaille les trois cas d'utilisation de la fonction `F` :

- `F()` ou `F(None,None)`, ce premier cas est sans doute le plus déroutant puisqu'il faut retourner le nombre de contraintes non linéaires et le premier x_0
- `F(x)` ou `F(x,None)`
- `F(x,z)`

L'algorithme de résolution est itératif : on part d'une point x_0 qu'on déplace dans les directions opposés aux gradients de la fonction à minimiser et des contraintes jusqu'à ce que le point x_t n'évolue plus. C'est pourquoi le premier d'utilisation de la focntion F est en fait une initialisation. L'algorithme d'optimisation a besoin d'un premier point x_0 dans le domaine de défintion de la fonction f.

[2]:
```
from cvxopt import solvers, matrix
import random

def fonction(x=None,z=None) :
    if x is None :
```

```
        x0 = matrix ( [[ random.random(), random.random() ]])
        return 0,x0
    f = x[0]**2 + x[1]**2 - x[0]*x[1] + x[1]
    d = matrix ( [ x[0]*2 - x[1], x[1]*2 - x[0] + 1 ] ).T
    if z is None:
        return f, d
    else :
        h = z[0] * matrix ( [ [ 2.0, -1.0], [-1.0, 2.0] ])
        return f, d, h

A = matrix([ [ 1.0, 2.0 ] ]).trans()
b = matrix ( [[ 1.0] ] )

sol = solvers.cp ( fonction, A = A, b = b)
print (sol)
print ("solution:",sol['x'].T)
```

```
     pcost       dcost       gap    pres   dres
 0:  0.0000e+00  4.3222e-01  1e+00  1e+00  1e+00
 1:  2.6022e-01  4.2857e-01  1e-02  1e-01  1e-02
 2:  4.2687e-01  4.2857e-01  1e-04  1e-03  1e-04
 3:  4.2855e-01  4.2857e-01  1e-06  1e-05  1e-06
 4:  4.2857e-01  4.2857e-01  1e-08  1e-07  1e-08
 5:  4.2857e-01  4.2857e-01  1e-10  1e-09  1e-10
Optimal solution found.
{'dual objective': 0.42857142857142855, 'primal objective': 0.4285714268720223,
'primal slack': 1.000000000000004e-10, 'snl': <0x1 matrix, tc='d'>, 'relative
gap': 2.333333333333341e-10, 'sl': <0x1 matrix, tc='d'>, 'dual slack':
0.9999999999999991, 'status': 'optimal', 'y': <1x1 matrix, tc='d'>, 'x': <2x1
matrix, tc='d'>, 'zl': <0x1 matrix, tc='d'>, 'znl': <0x1 matrix, tc='d'>, 'dual
infeasibility': 9.995026102717158e-11, 'gap': 1.0000000000000031e-10, 'primal
infeasibility': 1.2214984990086318e-09}
solution: [ 4.29e-01  2.86e-01]
```

1.2 Exercice 2 : l'algorithme de Arrow-Hurwicz

[3]:
```
def fonction(X) :
    x,y = X
    f = x**2 + y**2 - x*y + y
    d = [ x*2 - y, y*2 - x + 1  ]
    return f, d

def contrainte(X) :
    x,y = X
    f = x+2*y-1
    d = [ 1,2]
    return f, d

X0  = [ random.random(),random.random() ]
p0  = random.random()
epsilon = 0.1
rho     = 0.1
```

```
diff = 1
iter = 0
while diff > 1e-10 :
    f,d   = fonction( X0 )
    th,dt = contrainte( X0 )
    Xt    = [ X0[i] - epsilon*(d[i] + dt[i] * p0) for i in range(len(X0)) ]

    th,dt = contrainte( Xt )
    pt    = p0 + rho * th

    iter += 1
    diff = sum ( [ abs(Xt[i] - X0[i]) for i in range(len(X0)) ] )
    X0 = Xt
    p0 = pt
    if iter % 100 == 0 :
        print ("i {0} diff {1:0.000}".format(iter,diff),":", f,X0,p0,th)

print (diff,iter,p0)
print("solution:",X0)
```

```
i 100 diff 0.001 : 0.4247897864911925 [0.42689355878508073, 0.28435244828471445]
-0.5749941851846841 -0.004401544645490363
i 200 diff 4e-06 : 0.42858505488192533 [0.4285774472613499, 0.285720126646874]
-0.5714194393546865 1.7700555097865944e-05
i 300 diff 1e-08 : 0.42857138230973435 [0.4285714081852857, 0.28571426356925744]
-0.5714285933247762 -6.467619939609648e-08
8.65032490083e-11 353 -0.5714285707449258
solution: [0.42857142760818157, 0.28571428421894984]
```

La code proposé ici a été repris et modifié de façon à l'inclure dans une fonction qui s'adapte à n'importe quel type de fonction et contrainte dérivables : Arrow_Hurwicz. Il faut distinguer l'algorithme en lui-même et la preuve de sa convergence. Cet algorithme fonctionne sur une grande classe de fonctions mais sa convergence n'est assurée que lorsque les fonctions sont quadratiques.

1.3 Exercice 3 : le lagrangien augmenté

[4]:
```
def fonction(X,c) :
    x,y = X
    f = x**2 + y**2 - x*y + y
    d = [ x*2 - y, y*2 - x + 1 ]

    v = x+2*y-1
    v = c/2 * v**2

    # la fonction retourne maintenant dv (ce qu'elle ne faisait pas avant)
    dv = [ 2*(x+2*y-1), 4*(x+2*y-1) ]
    dv = [ c/2 * dv[0], c/2 * dv[1] ]
    return f + v, d, dv

def contrainte(X) :
    x,y = X
    f = x+2*y-1
    d = [ 1,2]
```

```
    return f, d

X0  = [ random.random(),random.random() ]
p0  = random.random()
epsilon = 0.1
rho     = 0.1
c       = 1

diff = 1
iter = 0
while diff > 1e-10 :
    f,d,dv = fonction( X0,c )
    th,dt = contrainte( X0 )
    # le dv[i] est nouveau
    Xt    = [ X0[i] - epsilon*(d[i] + dt[i] * p0 + dv[i]) for i in range(len(X0)) ]

    th,dt = contrainte( Xt )
    pt    = p0 + rho * th

    iter += 1
    diff = sum ( [ abs(Xt[i] - X0[i]) for i in range(len(X0)) ] )
    X0 = Xt
    p0 = pt
    if iter % 100 == 0 :
        print ("i {0} diff {1:0.000}".format(iter,diff),":", f,X0,p0,th)

print (diff,iter,p0)
print("solution:",X0)
```

```
i 100 diff 9e-06 : 0.4284835397042614 [0.4285257942646543, 0.28566702773052666]
-0.5712849940942254 -0.00014015027429237215
i 200 diff 8e-10 : 0.42857142064500353 [0.4285714244564873, 0.2857142814529341]
-0.5714285584819018 -1.263764448644622e-08
9.59901602648e-11 223 -0.5714285699086377
solution: [0.42857142808833615, 0.28571428521400477]
```

1.4 Prolongement 1 : inégalité

Le problème à résoudre est le suivant :
$\begin{cases} \min_U J(U) = u_1^2 + u_1^2 - u_1 u_2 + u_2 \\ \textit{sous contrainte } \theta(U) = u_1 + 2u_2 - 1 = 0 \textit{ et } u_1 \geqslant 0.3 \end{cases}$

[5]:
```
from cvxopt import solvers, matrix
import random

def fonction(x=None,z=None) :
    if x is None :
        x0 = matrix ( [[ random.random(), random.random() ]])
        return 0,x0
    f = x[0]**2 + x[1]**2 - x[0]*x[1] + x[1]
    d = matrix ( [ x[0]*2 - x[1], x[1]*2 - x[0] + 1 ] ).T
    h = matrix ( [ [ 2.0, -1.0], [-1.0, 2.0] ])
    if z is None: return  f, d
    else : return f, d, h
```

```
A = matrix([ [ 1.0, 2.0 ] ]).trans()
b = matrix ( [[ 1.0] ] )

G = matrix ( [[0.0, -1.0] ]).trans()
h = matrix ( [[ -0.3] ] )

sol = solvers.cp ( fonction, A = A, b = b, G=G, h=h)
print (sol)
print ("solution:",sol['x'].T)
```

```
     pcost       dcost       gap    pres   dres
 0:  0.0000e+00  5.1249e-01  2e+00  1e+00  9e-01
 1:  3.9825e-01  4.1007e-01  7e-02  4e-02  9e-03
 2:  4.3458e-01  4.2532e-01  1e-02  3e-03  2e-16
 3:  4.3118e-01  4.2927e-01  4e-03  9e-04  4e-16
 4:  4.3033e-01  4.2993e-01  8e-04  2e-04  5e-16
 5:  4.3005e-01  4.3000e-01  1e-04  2e-05  6e-16
 6:  4.3000e-01  4.3000e-01  3e-06  9e-07  1e-15
 7:  4.3000e-01  4.3000e-01  4e-08  1e-08  1e-15
Optimal solution found.
{'status': 'optimal', 'primal infeasibility': 9.636674935018775e-09, 'y': <1x1
matrix, tc='d'>, 'snl': <0x1 matrix, tc='d'>, 'sl': <1x1 matrix, tc='d'>,
'primal slack': 8.714188716285685e-11, 'dual slack': 0.20000259350365998,
'relative gap': 8.636668517465485e-08, 'znl': <0x1 matrix, tc='d'>, 'primal
objective': 0.43000001865573895, 'x': <2x1 matrix, tc='d'>, 'gap':
3.713767462508084e-08, 'dual objective': 0.4299999999997598, 'dual
infeasibility': 1.1909050386352223e-15, 'zl': <1x1 matrix, tc='d'>}
solution: [ 4.00e-01  3.00e-01]
```

1.5 Version avec l'algorithme de Arrow-Hurwicz

[6]:
```
import numpy,random

X0 = numpy.matrix ( [[ random.random(), random.random() ]]).transpose()
P0 = numpy.matrix ( [[ random.random(), random.random() ]]).transpose()

A  = numpy.matrix([ [ 1.0, 2.0 ], [ 0.0, -1.0]  ])
tA = A.transpose()
b = numpy.matrix ( [[ 1.0], [-0.30] ] )

epsilon = 0.1
rho     = 0.1
c       = 1

first = True
iter  = 0
while first or abs(J - oldJ) > 1e-8 :
    if first :
        J = X0[0,0]**2 + X0[1,0]**2 - X0[0,0]*X0[1,0] + X0[1,0]
        oldJ = J+1
        first = False
```

```
    else :
        oldJ = J
        J = X0[0,0]**2 + X0[1,0]**2 - X0[0,0]*X0[1,0] + X0[1,0]

    dj   = numpy.matrix ( [ X0[0,0]*2 - X0[1,0], X0[1,0]*2 - X0[0,0] + 1 ] ).
 ↪transpose()

    Xt = X0 - ( dj + tA * P0 ) * epsilon
    Pt = P0 + ( A * Xt - b) * rho

    if Pt [1,0] < 0 : Pt[1,0] = 0

    X0,P0 = Xt,Pt
    iter += 1
    if iter % 100 == 0 :
        print ("iteration",iter, J)

print (iter)
print ("solution:",Xt.T)
```

```
iteration 100 0.438928769823
iteration 200 0.431511185295
iteration 300 0.430474413611
iteration 400 0.430161042987
iteration 500 0.430060328795
iteration 600 0.430025004165
iteration 700 0.430011288151
iteration 800 0.430005417575
iteration 900 0.430002702711
988
solution: [[ 0.39982692  0.30007332]]
```

1.6 Prolongement 2 : optimisation d'une fonction linéaire

Correction à venir.

[7]: