Optimisation linéaire et convexité
TP5

23 mars 2016

Maxime Chupin, bureau 16-26-333, chupin@ann.jussieu.fr.

Dans cette séance nous allons implémenter quelques fonctions en python permettant de traiter de manière relativement générique les problèmes d'optimisation linéaire de première ou de seconde espèce.

Nous allons procéder par étapes, tout d'abord par la résolution de problèmes de première espèce, puis par celle de problèmes de deuxième espèce.

Rappelons le principe de fonctionnement d'une fonction python. Une fonction python se définit en suivant le schéma suivant.

```
def nomFonction(arg):
    ...
    return val
```

Par exemple, la fonction qui à x, un réel, renvoie $f(x) = x^2 + 4$, un autre réel, s'implémente de la façon suivante

```
def f(x):
   return x*x+4
```

On peut avoir plusieurs arguments d'entrée et de sortie. Par exemple la fonction suivante

```
def droite(X,Y):
   a=Y(2)-Y(1);
   b=X(1)-X(2);
   c=-(b*Y(1)+a*X(1));
   return a,b,c

# appel de la fonction définie
Y = np.array([2.5,1])
X = np.array([3,1])

a,b,c = droite(X,Y)
print a,b,c
```

Celle fonction calcule les coefficients (a, b, c) définissant la droite d'équation $ax + by + c = 0$ passant par les points `X` (vecteur à deux composantes) et `Y` (idem). Cette fonction a donc deux entrées (deux `np.array`) et trois sorties (trois scalaires).

Attaquons maintenant le TP en lui-même. On va bien entendu reprendre la fonction que l'on a utilisé jusque là : `pivot`.

```python
# la fonction pivot
def pivot(M,i,j):
    # M est une np.array (2 dimensions)
    # i indice de la ligne
    # indice de la colonne
    N = M.copy()
    l,c = np.shape(N)
    for k in np.arange(l):
        if k==i : # on est sur la ligne de i
            N[k,:] = M[k,:]/M[k,j] # on normalise la ligne pour avoir 1 en (i,j)
        else:
            # on soustrait aux lignes pour annuler au dessus et en dessous du pivot
            N[k,:]=M[k,:] - M[k,j]/M[i,j]*M[i,:]
    return N
```

1 Résolution des problèmes de première espèce

Exercice 1: Considérons un problème *de première espèce* sous sa forme *canonique*

$$\max Z(x) = c^T x \quad \begin{cases} Ax \leqslant b \\ x \geqslant 0 \end{cases} \tag{1}$$

1. Réaliser une fonction `standard` qui, à partir des seules données A, b et c, construit le tableau M final associé à la forme matricielle du problème mis sous forme standard. On utilisera les types `np.array`. La ligne associée au coût sera placée en dernière ligne. Le prototype de la fonction est le suivant :

```python
def standard(A,b,c):
    ...
    return M
```

2. Créer une fonction `dantzig` qui, à partir du tableau M du problème sous forme standard, renvoie la ligne et la colonne du pivot déterminé par la méthode de Dantzig. Le prototype de la fonction est le suivant

```python
def dantzig1(M):
    ...
    return ligne,colonne
```

   Les fonctions `np.max`, `np.amax`, `np.where`, `np.shape` peuvent être utiles.

3. En utilisant les trois fonctions `pivot`, `standard` et `dantzig`, construire une fonction `simplexePremiere` qui, à partir des données A, b et c, renvoie la matrice finale obtenue par méthode du simplexe (donc contenant la solution optimale si elle existe). Le prototype de la fonction est le suivant

```python
def simplexePremiere(A,b,c,NbMax):
    ...
    return M
```

   L'argument `NbMax` permet l'arrêt de la procédure si on dépasse `NbMax` pivots (détection de cyclages, etc.). La boucle `while` devra sans doute être utilisée. On pourra utiliser la fonction `np.max`.

4. Tester votre fonction sur les exemples précédemment traités en TP.

2 Résolution des problèmes de deuxième espèce

Exercice 2: On considère maintenant un problème de *deuxième espèce* sous forme *canonique* (noter le sens de l'inégalité).

$$\max Z(x) = c^T x \quad \begin{cases} Ax \leqslant b \\ x \geqslant 0 \end{cases} \tag{2}$$

Ici, par définition, b a au moins une composante strictement négative.

1. Réaliser une fonction `standardDeuxieme` qui à partir de A, b et c, renvoie le tableau récapitulatif du problème auxiliaire et du problème d'optimisation linéaire (problème auxiliaire permettant d'obtenir une solution de base réalisable pour le problème initial). Il faudra, dans cette fonction, ajouter la variable auxiliaire dans la base de solution.
   Le prototype de la fonction sera le même que celui de la fonction `standart`, à savoir

```
def standartDeuxieme(A,b,c)
    ...
    return M
```

2. Modifier la fonction `dantzigDeuxieme` pour pouvoir spécifier la ligne de coût (dernière ou avant dernière) à partir de laquelle le critère de DANTZIG sera effectué. Le prototype de la fonction sera alors

```
def dantzigDeuxieme(M,cout):
    ...
    return ligne, colonne
```

   Cette fonction servira donc dans les deux phases de la résolution du problème. Suivant la valeur de `cout`, on utilisera la ligne de coût relative au problème auxiliaire (`cout==1`) ou au problème initial (`cout==2`).
3. À partir des fonctions `standardDeuxieme` et `dantzigDeuxieme`, réaliser une fonction `simplexeDeuxieme` qui, à partir de A, b, et c, résout le problème d'optimisation linéaire en deux phases, dans un premier temps la résolution du problème auxiliaire puis celle du problème initial à partir de la solution de base réalisable obtenue. Le prototype de la fonction sera le même que précédemment

```
def simplexeDeuxieme(A,b,c,NbMax):
    ...
    return M
```

4. Tester la fonction sur les exemples déjà traités dans les précédents TPs.

3 Pour aller plus loin

Exercice 3: Pour améliorer l'ensemble, on pourra implémenter le critère de BLAND. On pourra aussi donner la solution sous forme de vecteur. On pourra faire une fonction qui teste s'il s'agit d'un problème de première ou deuxième espèce. On pourra enfin résoudre le problème dual.