ENSTA **Année scolaire 2015-2016**

A9-1. Outils d'optimisation.

Aide-mémoire pour le langage de modélisation AMPL

---

1 Structure générale

En tant que logiciel AMPL se présente sous forme d'une console interprétant les commandes de l'utilisateur. Les commandes AMPL se divisent en deux grandes catégories :

- Commandes permettant de décrire un problème d'optimisation à résoudre ; la descritpion d'un problème d'optimisation eut être enregistrée dans un fichier texte et chargée dans la console à l'aide d'une commande appropriée (un peu comme pour Matlab ou Scilab)
- Commande permettant de paramétrer le fonctionnement d'AMPl, de résoudre un problème donné à l'aide d'un solveur et de visualiser ou enregistrer les résultats

2 Console AMPL

La ligne de commande de la console AMPL commence par

```
ampl:
```

Si une précédente commande a été incomplète (par exemple, il manque ';', obligatoire à la fin de chaque commande) la console est en attente d'un complément :

```
ampl?
```

En cas d'erreurs d syntaxe dans les commandes (entrées directement en ligne ou chargées depuis un fichier) la console affiche les messages d'erreur :

```
ampl: var {1:4} y

syntax error
context:  var  >>> { <<< 1:4} y
ampl? var {1..4} y
ampl? ;
```

Le fonctionnement de la console peut être paramétré à l'aid de la commande

```
option NOM_OPTION VALEUR_OPTION;
```

Par exemple, pour préciser le solveur à utiliser :

```
option solver snopt;
```

ou

```
option solver minos;
```

Pour indiquer le chemin d'accès aux solveurs et aux fichiers de modèles :

```
option PATH '.';
```

Si les solveurs se trouvent dans le même répertoire que l'exécutable d'AMPL.

Pour charger et résoudre un problème on utilise les commandes de console suivantes :

— `model` : permet de charger un problème depuis un fichier. Exemple :

```
model modeleR.mod;
```

permet de charger les instructions contenues dans le fichier `modeleR.mod`.

— `solve` : permet de résoudre le problème chargé. Tapez :

```
solve;
```

— `display` : permet d'afficher les valeurs des variables ou paramètres. Exemple :

```
display x,y;
```

permet d'afficher les valeurs des variables de décision x et y, puis

```
display J_Rosen;
```

affiche la valeur de la fonction objectif au point d'optimum trouvé.

— `reset` : remet à zéro la mémoire de la console ;

— `quit` : permet de quitter la console.

3 Décrire un problème d'optimisation

3.1 Commandes de base

Pour décrire un problème de minimisation de type $\min_{x\in K} J(x)$ on doit :

— définir les variables de décision : mot-clé `var`

```
var nom1;
var nom2;
...
```

— définir la fonction objectif et le type d'optimisation :

```
minimize NomFonction: expression;
```

— décrire l'ensemble de contraintes K : mot-clé `s.t.`

```
s.t. nom1: expression1;
s.t. nom2: expression2;
...
```

— Donner une valeur initiale pour l'approximation de la solution : mot–clé `let`

```
let var1:=expression1;
let var2:=expression2;
...
```

— toutes les commandes doivent se terminer par " ;"

Exemple. Considérons le problème de minimisation de la fonction de Rosenbrock

$$\min_{x,y\in\mathbf{R}^2}(1-x)^2+100(y-x^2)^2$$

Son minimum global est $x_0=(1,1)$.

Ce problème de minimisation se définit avec AMPL comme suit :

```
 #   variables de décision
var x;
var y;

# Définition de la fonction objectif
minimize J_Rosen: 100*(y - x^2)^2 + (1-x)^2;

# Donner une valeur initiale pour l'algorithme (optionnel)
let x:=-1.;
let y:=1.1;
```

Un problème d'optimisation peut être enregistré dans un fichier sous forme de texte.

3.2 Travailler avec des quantités vectorielles

Pour déclarer une variable de décision sous forme de tableau (vecteur) on utilise la syntaxe

```
var nomVariable{débutIndice..finIndice};
```

Les indice peuvent prendre des valeurs négatives et positives. Par exemple, le code :

```
var x{0..5};
var y{-3..3};
```

déclare un vecteur x de $\mathbf{R}^6$ et un vecteur y de $\mathbf{R}^7$. On peut accéder aux composantes de x avec

```
x[0],  x[1], x[2], x[3], x[4], x[5]
```

On peut de même définir une matrice

```
var matrice{0..3, -3..3}
```

et accéder à ses éléments `matrice[0,-2]`.

Certaines opérations sur les vecteurs ont une syntaxe simplifiée :

```
sum {j in 1..4} x[j]
```

pour une somme et `prod {j in 1..4} x[j]` pour un produit.

IMPORTANT ! Pour répéter une expression dépendant d'indice on insère devant

```
{indice in début..fin}
```

Par exemple, on peut **définir plusieurs contraintes dépendant d'un indice**

```
s.t. contraintesLin {j in 1..3}: sum{i in 1..3} (i+j)*x[j] <=2*j
```

définit la contrainte

$$\sum_{i=1}^{3}(i+j)x_j <= 2*j, \quad j=1,2,3$$

On peut **attribuer une valeur initiale à une variable vectorielle** de la façon suivante

```
let {j in 1..3}: x[j]:=j;
```

3.3 Déclaration de paramètres

Le mot clé `param` permet de définir des paramètres du problème, des coefficients constants par exemple. On peut leur attribuer une valeur à la déclaration

```
param n:=4;
```

et utiliser leur nom plus loin :

```
var x{1..n}>=0;
```

On peut définir une matrice paramètre

```
param a: 1 2 3 4 :=
1 4 2.25 1 0.25
2 0.16 0.36 0.64 0.64
;
```

ou de manière équivalente

```
param a:=
1 1 4
1 2 2.25
1 3 1
1 4 0.25
2 1 0.16
2 2 0.36
2 3 0.64
2 4 0.64
;
```

3.4 Définir des quantités dépendant des variables de décision

Dans des modèles complexes, on peut avoir besoin de définir des quantités qui ne sont pas des fonctions de variables de décision. Pour les nommer et les utiliser dans tout le modèle on utilise le mot clé `var` avec une syntaxe particulière :

```
var nomFunc=expression;
```

ATTENTION ! Ne pas confondre le signe d'égalité utilisé ici avec le signe d'affectation `:=` ! Il s'agit ici d'une égalité "logique" et non d'une affectation. L'expression peut faire appel aux paramètres, variables de décision ou autres quantité déjà définies lus haut dans le modèle. Exemple :

```
param N=50; # paramètre définissant la taille du vecteur de variables de décision
var y{0..N}; # variables de décision
var yPrime {j in 1..N} =(y[j]-y[j-1])*N; # dérivée
```

3.5 Fonctions mathématiques

```
 abs(x)     sin(x)    cos(x)          exp(x)
log(x)     log10(x)  min(x,y,...)   max(x,y,...)
round(x,n) round(x)  precision(x,n) trunc(x)
trunc(x,n) sqrt(x)   floor(x)
```

et quelques fonctions spéciales

```
Beta(a,b)   Cauchy()   Exponential()   Gamma(a)
Irand224()  Normal()   Poisson()       Uniform(m,n)
```

Le nombre π

```
param pi := 4*atan(1);
```

3.6 Ecrire les résultats dans un fichier

```
printf [index:] "format",listeVariables
> nomFichier;
```

Exemple :

```
ampl: printf {i in 0..N-1}: "%10f %10f \n",i*h, v[i]
ampl? > vitesse.dat;
```