Modéliser avec AMPL

6 février 2002

Introduction

AMPL est un logiciel qui permet de formuler mathématiquement des problèmes d'optimisation. Le modèle est décrit par l'utilisateur dans un langage humainement compréhensible. La description de ce langage est l'objet de ces quelques pages. Le rôle d'AMPL est de générer une autre représentation équivalente mais facilement compréhensible par *l'algorithme* qui résoudra le problème. Cette dernière représentation est typiquement extensive (par exemple la matrice pour un problème linéaire).

1 Généralités.

AMPL ignore tous les blancs, les retours a la ligne,.... C'est le caractère ; qui indique la fin d'une expression/définition, pas le retour à la ligne.

Les noms des entités (variables, contraintes, paramètres,...) peuvent inclure des lettres majuscules ou minuscules (considérées comme différentes) et/ou des nombres.

2 Constantes.

Les constantes numériques s'écrivent de la manière habituelle :

Exemple : 0.0123, 1.23D-2, 1.23e-2, 1.23E-2
sont toutes des déclarations valides et équivalentes.

Les littéraux sont delimités par ' ou par ".

Exemple : 'abc','x','y'.

Note la constante 1 est différente du littéral '1'.

Le commentaires comencent par # jusqu'à la fin de la ligne ou peuvent commecer par /* et finir par */ en utilisant plusieurs lignes.

3 Ensembles.

Un ensemble est un groupement de zéro ou plusieurs éléments distincts, mais de même type. En particulier, tous les éléments doivent avoir le même nombre de composantes (tous des paires, des triplets,...). La definition d'un ensemble comence par { et fini par }.

Exemple :
{1,2,3,4,5,6,7,8,9,10}
{"a","b","c","d"}
{"janvier","fevrier","mars","avril","mai","juin"}
{(1,2,3),(4,5,5),(6,7,8),(9,10,1)}
{("janvier",1),("fevrier",2),("mars",3),("avril",4) ,("mai",5),("juin",6)}

Des ensembles d'entiers peuvent être déclarés de manière concise :

Exemple :
1990..2020 by 5
est équivalent à
{1990, 1995, 2000, 2005, 2010, 2015, 2020}.

4 Indices.

Les ensembles sont surtout utilisés pour indexer des tables, mais également pour les sommations complexes et les boucles. Les exemples suivants permettent de comprendre la manière dont les indices sont traités par AMPL.

{A}	#	un ensemble.
{A,B}	#	tous le pairs (a, b) : $a \in A, b \in B$.
{i in A,j in B}	#	idem.
{i in A, B}	#	idem.
{i in A, C[i]}	#	toutes le paires (a, b) : $a \in A, b \in C[i]$.
{i in A, (j,k) in D}	#	toutes le paires $(i, (j, k))$: $i \in A, (j, k) \in D$.
{i in A : p[i] > 0}	#	toutes les $a \in A$ tels que $p[i] > 0$.
{i in A,j in C[i] : i <= j}	#	dans ce cas i et j doivent être des nombres.
{i in A, (i,j) in D : i <= j}	#	

En général on a :

indices
{ setxpr-list }
{ setxpr-list : lexpr }

sexpr-list
setxpr
object in setxpr
setxpr-list, setxpr-list

5 Expressions arithmétiques et logiques.

- Une constante, un symbole ou un élément d'un tableau sont des expressions arithmétiques.
- Si *exp* est une expression arithmétique ne commençant pas par -, - *exp* l'est aussi.
- Si *exp* est une expression arithmétique, (*exp*) l'est aussi.
- Si exp_1 et exp_2 sont des expressions arithmétiques, alors $exp_1 + exp_2$, $exp_1 - exp_2$, $exp_1 * exp_2$, exp_1/exp_2 et $exp_1\hat{}exp_2$ le sont aussi.
- $build - in\ exp$ est aussi une expression ou $build - in$ est une fonction d'AMPL.
- Si *lexpr* est une expression logique et *vexpr* est une expression, alors ceci est une expression : if *lexpr* then *vxpre* [else vexpr].

– -Infinity et Infinity sont des expressions valides.

La liste complete des opérateurs :

Nom	Type	Remarque	Nom	Type	Remarque
if then else	A,S		or ou | |	L	
exist forall	L	opérateurs logiques	and ou &&	L	
< <= > >= <> ! =	L		in - not in	L	appartenir à un ensemble
within - not within	L	S within T signifie $S \subseteq T$	not -!	L	
union - diff -symdiff	S		inter	S	
cross	S	produit cartésien	set of .. by	S	
+ - less	A		sum - prod min - max	A	
- / - div - mod	A		+ -	A	opérateur unitaire
^- **	A	exposant			

6 Fonctions pré-définies.

Ces fonctions sont déjà incluses dans AMPL. Les plus courantes sont :

```
abs(x)       ceil(x)    floor(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)
```

Toutes les fonctions trigonométriques et hyperboliques sont également définies. Pour finir, certaines fonctions permettent de générer des nombres aléatoires.

```
Beta(a,b)      Cauchy()      Exponential()   Gamma(a)
Irand224()     Normal(μ,σ)   Normal01()      Poisson(μ)
Uniform(m,n)   Uniform01()
```

7 Déclaration des entités d'un modèle.

Les entités sont déclarées de la manière suivante :

entité nom [alias] [index] [corps];

où, *nom* est une expression alphanumérique qui n'a pas été utilisée auparavant, *alias* et *index* sont des expressiones optionelles. Finalement, *entité* est un des mots-clef :

```
set
param
var
arc
minimize
maximize
subject to
node
```

Si l'*entité* est omise, AMPL suppose `subject to` par défaut.

Exemple :

```
set nodes ;
set arcs within nodes cross nodes ;
param max_iter := card(nodes)-1 ; # card(s) = number of elements in s.
```

Les déclaration des entités peuvent apparaître dans un ordre quelconque. Dans les sections qui suivent nous allons voir comment déclarer chaque type d'entité.

7.1 Déclaration de `set`

La déclaration d'un ensemble se fait selon le format général suivant :

`set` *nom [alias] [index] [attribs]*;

ou *attribs* est une liste des attributs de l'ensemble :

attribs :

dimen n,	dimension de l'ensemble.
within sexpr,	l'ensemble defini est un sousensemble de *sexpr*
:= sexpr,	definition de l'ensemble
default sexpr ;	definition par défault de l'ensemble.

Note : `default` et `:=` sont mutuellement exclusifs. `default` est utilisé seulement si l'ensemble n'est pas défini par les déclaration des éléments de l'ensemble a la même ligne.

Exemple :

```
set nodes ; # il doit être défini dans la section data.
set step {s in 1..maxiter } dimen 2 :=
if s == 1 then arcs
else step[s-1] union setof {k in nodes,
(i,k) in step[s-1], (k,j) in step[s-1]} (i,j) ;
set A := 1..n ;
set C := A union B ; # A et B sont deux ensembles.
set C := A diff B ;
set C := A symdiff B ;
```

Note Si `A` est un ensemble, `card(A)` est le cardinal de l'ensemble.
Il est possible définir deux types d'ensembles particuliers :
`interval[a,b]` # $= \{x : a \leq x \leq b\}$
`interval(a,b)` # $= \{x : a < x < b\}$
`interval(a,b]` # $= \{x : a < x \leq b\}$
`integer[a,b]` # $= \{x \in I\!N : a \leq x \leq b\}$
`integer(a,b)` # $= \{x \in I\!N : a < x < b\}$
`integer(a,b]` # $= \{x \in I\!N : a < x \leq b\}$

7.2 Déclaration de `parameter`

On déclare un paramètre selon le format général suivant :

`param` *nom [alias] [index] [attribs]*;

attribs :

binary,	le paramètre doit être binaire,
integer,	le paramètre doit être entier,
symbolic,	le paramètre peut prendre une valeur alphanumérique (par exemple le nom d'un élément d'un autre ensemble).
relop expr,	où *relop* peut être < <= = == ! = <> > >=.
in setxpr,	vérifie si le paramètre appartient à un ensemble donné.
:= expr,	définition du paramètre.
default expr	même chose que pour la déclaration d'un ensemble.

Exemple :

```
param units {raw,prd} >= 0;   # units[i,j] is the quantity of raw material i
                              # needed to manufacture one unit of product j

param profit {prd,1..T};      # profit[j,t] is the estimated value (if >= 0)
                              # or disposal cost (if <= 0) of
                              # a unit of product j in period t

param comb 'n choose k'  \{n in 0..N, k in 0..N \}
     := if k = 0 or k = N then 1 else comb[n-1,k-1] + comb[n-1,k];

param dem 'demand' {prd,first..last+1} >= 0;
    # Requirements (in 1000s) to be met from current production and inventory
```

7.3 Déclaration des variables.

Le format général de déclaration des variables est le suivant :

`var` *nom [alias] [index] [attribs]*;

attribs :

binary,	la variable est binaire
integer,	la variable est entière
>= *expr*,	si la variable est inférieurement bornée
<= *expr*,	si la variable est supérieurement bornée
:= expr,	valeur d'une solution initiale (information utilisé par certains algorithmes)
default expr	valeur par défaut d'une solution initiale
= vexpr,	fixe la variable à une borne
coeff [index] constraint vexpr,	pour la d'efinition par colonne d'un modèle
cover [index] constraint,	pour la d'efinition par colonne d'un modèle
obj [index] objective vexpr,	pour la d'efinition par colonne d'un modèle

Exemple :

```
var Make {p in PROD} >= commit[p], <= market[p]; # tons produced

var Make {prd,1..T} >= 0;     # Make[j,t] is the number of units of product j
                              # manufactured in period t

var Store {raw,1..T+1} >= 0;  # Store[i,t] is the number of units of raw
                              # material i in storage at the beginning
                              # of period t

var Rprd 'regular production' {prd,time} >= 0;
                        # Production using regular-time labor, in 1000s
```

7.4 Déclaration de contraintes

déclaration de une contrainte :
[`subject to`] *name* [alias] [index]
[:= *initial_dual*] [`default` *initial_dual*] [: contraint_expression] ;

Exemple :

```
subject to limit {t in 1..T}:  sum {j in prd} Make[j,t] <= max_prd;
            # Total production in each period must not exceed maximum


subject to balance {i in raw, t in 1..T}:
     Store[i,t+1] = Store[i,t] - sum {j in prd} units[i,j] * Make[j,t];


rlim 'regular-time limit' {t in time}:
    sum {p in prd} pt[p] * Rprd[p,t] <= sl * dpp[t] * Crews[t];




dreq 'demand requirements' {p in prd, t in first+1..last}:
    Rprd[p,t] + Oprd[p,t] + Short[p,t] - Short[p,t-1]
  + sum {a in 1..life} (Inv[p,t-1,a] - Inv[p,t,a])
                          = dem[p,t] less iil[p,t-1];

                                # Production plus increase in shortage plus
                                # decrease in inventory must equal demand
```

7.5 Déclaration de l'objectif

déclaration de l'objectif :
`maximize` *name* [alias] [index] : expression ; `minimize` *name* [alias] [index] : expression ;

expression :
vexpr
to_come + vexpr
vexpr + to_come
to_come

Exemple :

```
maximize total_profit: sum {p in PROD} profit[p] * Make[p];
              # Objective: total profits from all products

maximize total_profit:
      sum {t in 1..T} ( sum {j in prd} profit[j,t] * Make[j,t] -
                        sum {i in raw} cost[i] * Store[i,t] )
    + sum {i in raw} value[i] * Store[i,T+1];

                  # Total over all periods of estimated profit,
```

```
                    # minus total over all periods of storage cost,
                    # plus value of remaining raw materials after last period

minimize cost:  sum {p in prod, f in fact} rpc[p,f] * Rprd[p,f] +
sum {p in prod, f in fact} opc[p,f] * Oprd[p,f] +
sum {p in prod, (d,w) in rt} sc[d,w] * wt[p] * Ship[p,d,w] +
sum {p in prod, d in dctr} tc[p] * Trans[p,d];

# Total cost:  regular production, overtime
# production, shipping, and transshipment
```

7.6 Les Suffixes des variables.

Chaque variable, contrainte ou objectif possède diverses valeurs associées. Par exemple, à une variable sont associées des bornes, un coût réduit tandis qu'à une contrainte sont associées une variable duale et une variable d'écart. Toutes ces valeurs sont accessibles par le biais d'un suffixe associé, du type *nom*.suff. Quelques suffixes utiles se trouvent dans le tableau suivant :

Variable		Contrainte		Objectif	
suffixed	descrip.	suffixe	descrip	suffixe	descrip
.lb	borne inférieure	.dual	valuer dual actuel	.val	valeur
.ub	borne supérieure	.slack	valuer actuel de l'écart		
.rc	coût reduit				
.val	valeur courant				

8 Définition des données.

La définition des données se fait habituellement dans un fichier distinct de celui qui définit le modéle. Cela permet de faire la distinction entre une instance particulière et le modèle générique. L'introduction de données obéit d'ailleurs à des règles légèrement différentes de celles effectives lors de la définition du modèle. On débute donc toujours cette section par le mot-clef : `data ;`. Selon le type de données que l'on veut inclure, différents formats d'écriture sont possibles.

8.1 ensemble

La déclaration d'un ensemble est constituée du mot-clef `set`, le nom, facultativement :=, et les éléments.

Exemple :

```
data ;
set S := 'a' ''b'' c ;
set A := (1,2) (2,3) (4,2) (3,1) (2,2) (4,4) (3,4) ;
ou
set A :=
 :  1  2  3  4   :=
 1  -  +  -  -
 2  -  +  +  -
 3  +  -  -  +
 4  -  +  -  +   ;
set B := (1,2,3) (1,3,2) (2,3,1) (2,1,3) (1,2,2) (1,1,1) (2,1,1) ;
```

```
ou
set B :=
 (1,*,*)   :  1  2  3   :=
           1  +  -  -
           2  -  +  +
           3  -  +  -
 (2,*,*)   :  1  2  3   :=
           1  +  -  +
           2  -  -  -
           3  +  -  -   ;
ou
set B :=
(1,*,2) 3 2
(2,*,1) 3 1
(1,2,3) (2,1,3) (1,1,1)
ou
set B :=
(*,*,*) 1 2 3 1 3 2 2 3 1 2 1 3
1 2 2 1 1 1 2 1 1 ;
```

Dans le deuxième exemple, un "+" indique que l'élément (i,j) appartient à l'ensemble.
Pour les ensembles indexés (ensembles d'ensembles), chaque déclaration doit être faite séparément.

Exemple :

```
set PROD ;
set AREA { PROD } ;
set PROD := bands coils ;
set AREA[bands] := east north ;
set AREA[coils] := east west export ;
```

8.2 paramètre

La déclaration des paramètres est semblable à celle des ensembles. On l'expliquera avec des examples.

Exemple :

```
param T := 4 ; # T = 4
```

Si le paramètre est indexé par un ensemble, deux manières de procéder sont possibles :

Exemple :

```
param init_stock := iron 7.32 nickel 35.8 ;
ou
 param :  init_stock  cost  value   :=
   iron       7.32     .025   -.1
  nickel      35.8      .03    .02    ;
```

Les paramètres que l'on définit ainsi sont : init_stock[iron], init_stock[nickel], cost[iron], etc.

On peut aussi définir un ensemble implicitement par la déclaration d'un paramètre. L'ensemble ainsi défini est l'ensemble qui indexe le paramètre en question. Dans l'example précédent, si l'on veut définir l'ensemble des matières premières "raw", il suffit d'ajouter `:raw :` après le mot `param`.

```
param   :raw :  init_stock  cost  value   :=
         iron       7.32    .025   -.1
        nickel      35.8     .03    .02    ;
```

On déclare des paramètres à plusieurs dimensions comme pour les ensembles. Si le mot `(tr)` apparaît après le nom du paramètre, cela indique que la matrice est transposée (les colonnes sont les lignes et vice versa).

Exemple :

```
param demand (tr):
            FRA    DET    LAN    WIN    STL    FRE    LAF :=
    bands   300    300    100     75    650    225    250
    coils   500    750    400    250    950    850    500
    plate   100    100      0     50    200    100    250 ;
```

Si le paramètre a trois composantes :

Exemple :

```
param trans_cost :=

 [*,*,bands]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
       GARY     30   10    8   10   11   71    6
       CLEV     22    7   10    7   21   82   13
       PITT     19   11   12   10   25   83   15

 [*,*,coils]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
       GARY     39   14   11   14   16   82    8
       CLEV     27    9   12    9   26   95   17
       PITT     24   14   17   13   28   99   20

 [*,*,plate]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
       GARY     41   15   12   16   17   86    8
       CLEV     29    9   13    9   28   99   18
       PITT     26   14   17   13   31  104   20 ;
```

9 Les commandes

Quand vous exécutez AMPL, vous entrez dans un environnement interactif où les commandes suivantes sont possibles.

model, data Pour charger le modèle contenu dans le fichier `mymodel.mod` :
`ampl : model ; include mymodel.mod` ou `ampl : model mymodel.mod ;`

Pour charger les données contenu dans le fichier `mydata.dat` :
`ampl : data ; include mydata.dat` ou `ampl : data mydata.dat ;`

Exemple :

```
[core4ux] ampl

   ampl: model diet2.mod;
   ampl: data diet2.dat;
```

print, display, printf Ces commandes permettent d'imprimer la solution du problème courant et d'autres informations. La syntaxe générale est :

`display` index : [disp_list] [redirecc] ;
`print` index : arg_list [redirecc] ;
`printf` index : fmt, arg_list [redirecc] ;

Exemple :

```
[10:~/AMPL/]%ampl
ampl: model manuf.mod ;
ampl: data manuf1.dat ;
ampl: display T;
T = 4

ampl: display units;
units :=
iron   bolts     0.83
iron   nuts      0.79
iron   washers   0.92
nickel bolts     0.17
nickel nuts      0.21
nickel washers   0.08
;

ampl: display cost;
cost [*] :=
  iron  0.03
nickel  0.025
;
```

Après avoir trouvé la solution optimale :

```
ampl: display Make ;
Make :=
bolts   1    0
bolts   2    0
bolts   3    0
bolts   4   43.0044
nuts    1    0
nuts    2    0
nuts    3    0
nuts    4    0
```

```
washers 1    0
washers 2    0
washers 3    0
washers 4    0.115556
;
```

Vous pouvez également évaluer des expressions complexes.

```
ampl: display  sum {t in 1..T} ( sum {j in prd} profit[j,t] * Make[j,t] -
ampl?  sum {i in raw} cost[i] * Store[i,t] )
ampl?  + sum {i in raw} value[i] * Store[i,T+1];

sum{t in 1 .. T} (sum{j in prd} profit[j,t]*Make[j,t] - sum{i in raw} cost[i]*
  Store[i,t]) + sum{i in raw} value[i]*Store[i,T + 1] =
102.637

ampl: display Make,profit,Store,cost;
:             Make    profit   Store    :=
bolts   1    0          1.82     .
bolts   2    0          1.9      .
bolts   3    0          1.7      .
bolts   4   43.0044     2.5      .
iron    1       .         .    35.8
iron    2       .         .    35.8
iron    3       .         .    35.8
iron    4       .         .    35.8
iron    5       .         .     0
nickel  1       .         .     7.32
nickel  2       .         .     7.32
nickel  3       .         .     7.32
nickel  4       .         .     7.32
nickel  5       .         .     0
nuts    1    0          1.73     .
nuts    2    0          1.8      .
nuts    3    0          1.6      .
nuts    4    0          2.2      .
washers 1    0          1.05     .
washers 2    0          1.1      .
washers 3    0          0.95     .
washers 4    0.115556   1.33     .
;

cost [*] :=
  iron  0.03
nickel  0.025
;

ampl: display {t in 1..T} (  sum {j in prd} profit[j,t] * Make[j,t] ),
ampl?  {t in 1..T} (sum {i in raw} cost[i] * Store[i,t] ),
ampl? {t in 1..T} (  sum {j in prd} profit[j,t] * Make[j,t] -
ampl?  sum {i in raw} cost[i] * Store[i,t] );

# $1 = sum{j in prd} profit[j,t]*Make[j,t]
```

```
# $3 = sum{j in prd} profit[j,t]*Make[j,t] - sum{i in raw} cost[i]*Store[i,t]
:     $1    sum{i in raw} cost[i]*Store[i,t]     $3       :=
1     0                   1.257                -1.257
2     0                   1.257                -1.257
3     0                   1.257                -1.257
4   107.665               1.257               106.408
;
```

La commande `print` possède le même syntaxe que `display` mais la sortie n'est pas formatté.

Exemple :

```
ampl: print  {t in 1..T} (  sum {j in prd} profit[j,t] * Make[j,t] ),
ampl?  {t in 1..T} (sum {i in raw} cost[i] * Store[i,t] ),
ampl? {t in 1..T} (  sum {j in prd} profit[j,t] * Make[j,t] -
ampl?  sum {i in raw} cost[i] * Store[i,t] );
0 0 0 107.66480000000001 1.257 1.257 1.257 1.257 -1.257 -1.257 -1.257 106.407800
00000001
ampl:
```

La commande `printf` est identique à `print`, mais offre la possibilité de formatter la sortie comme en C.

Exemple :

```
ampl:  printf {t in 1..T}: "periode %d profit %.3f cout %.3f net %.3f \n",t,
ampl?  sum {j in prd} profit[j,t] * Make[j,t], sum {i in raw} cost[i] * Store[i,t],
ampl? sum {j in prd} profit[j,t] * Make[j,t] - sum {i in raw} cost[i] * Store[i,t] ;

periode 1 profit 0.000 cout 1.257 net -1.257
periode 2 profit 0.000 cout 1.257 net -1.257
periode 3 profit 0.000 cout 1.257 net -1.257
periode 4 profit 107.665 cout 1.257 net 106.408
ampl:
```

option Il existe plusieurs options qui peuvent être modifiées par l'utilisateur soit pour la visualisation de la solution, soit pour la stratégie de résolution. En voici la syntaxe, ainsi qu'une liste des options les plus utiles (`option` * ; liste toutes les options possibles).

`option` *envname envvalue* ;

`solver`	Le nom du solveur utilisé.
`display_eps`	Si la valeur est inférieure a `display_eps`, la valeur visualisée est zéro.
`objective_precision`	`omit_zero_cols` (<> omit)
`omit_zero_rows`	`output_precision`

solve Cette commande permet de résoudre le problème chargé en memoire au moyen de l'algorithme indiqué par l'option `solver`.

Exemple :

```
[12:~/AMPL/]%ampl
ampl:  model manuf.mod; data manuf2.dat ;
ampl: solve ;
CPLEX 6.0: Tried aggregator 1 time.
Reduced LP has 6 rows, 16 columns, and 40 nonzeros.

Iteration log . . .
Iteration:     1    Objective     =             0.000000
CPLEX 6.0: optimal solution; objective 851.8190433
6 iterations (0 in phase I)
ampl: option display_1col 1
ampl? ;
ampl: display Make ;
Make [*,*] (tr)
:     bolts   nuts   washers    :=
1   123.7       0      0
2   123.7       0      0
3     5.61111   0    114.489
4   123.7       0      0
;

ampl: quit
[13:~/AMPL/]%
```

solution La syntaxe est la suivante :

`solution` *filename* ;

Cette commande permet lire la valeur des variables primales et duales contenue dans le fichier *filename*. Le fichier doit être créé durant l'exécution de la commande `solve`.

write La commande `write` permet d'écrire de l'information dans un fichier. La syntaxe est :

`write` clef *filename* ;

objective,drop, restore Ces commandes ont la syntaxe suivante :

`drop` [index] const-or-objname ;

Permet l'élimination temporaire d'une contrainte ou d'un ensemble de contraintes.

`restore` [index] const-or-objname ;

Cette commande annule une commande `drop`.

`objective` objective_name ;

Permet de sélectionner un objectif s'il en existe plusieurs.

fix, unfix Ces commandes permettent de fixer une ou plusieurs variables. La syntaxe :

`fix` [index] var-name ;

`unfix` [index] var-name ;

shell Cette commande permet d'exécuter une commande du shell (en unix).

`shell` 'command-line' ;

`shell` ;

reset Pour retourner aux valeurs par défaut il suffit d'utiliser la commande `reset` dans un des formats suivants :

```
reset ;
reset options ;
reset data [name-list] ;
```

quit, end Pour quitter AMPL il suffit de taper `quit` ou `end`.

10 Les mots réservés.

Les mots suivants sont réservés ; ils ne peuvent donc pas être utilisés pour la définition d'une entité du modèle.

```
Infinity  data     else    if       less  setof     then
binary    default  exists  in       max   sum       union
by        dimen    forall  integer  min   symbolic  within
```

11 Running in Batch

Vous pouvez aussi utiliser AMPL en mode *batch*. Pour cela il faut créer un fichier avec toutes les commandes que l'on veut exécuter.

: fichier : example.run

```
model prod.mod ;
data prod13.dat ;
solve ;
option display_1col 5 ;
option display_round 1;
display Inv > inventory.sol ;
```

Si l'on exécute :

```
[54:~/AMPL/]%ampl example.run
CPLEX 6.0: Tried aggregator 1 time.
LP Presolve eliminated 5 rows and 5 columns.
Reduced LP has 637 rows, 861 columns, and 2706 nonzeros.

Iteration log . . .
Iteration:     1    Scaled infeas =          4988.686675
Iteration:   105    Scaled infeas =          2005.302400
Iteration:   211    Scaled infeas =            93.380041
Switched to devex.
Iteration:   257    Objective     =       3709023.350286
Iteration:   350    Objective     =       2438400.769325
Iteration:   421    Objective     =       2360771.146652
CPLEX 6.0: optimal solution; objective 2359313.555
482 iterations (256 in phase I)
[55:~/AMPL/]%
```

De plus, le fichier "inventory.sol" contiendra la valeur optimale de la variable Inv.

12 Exemples.

Les exemples suivant incluent le modèle et les données. Dans le premier exemple contient la résolution du problème.

12.1 A simple probème de fabrication.

```
# ----------------------
# SETS
# ----------------------

set prd;                        # products
set raw;                        # raw materials

# ----------------------
# PARAMETERS
# ----------------------

param T > 0 integer;            # number of production periods

param max_prd > 0;              # maximum units of production per period

param units {raw,prd} >= 0;     # units[i,j] is the quantity of raw material i
                                # needed to manufacture one unit of product j

param init_stock {raw} >= 0;    # init_stock[i] is the maximum initial stock
                                # of raw material i

param profit {prd,1..T};        # profit[j,t] is the estimated value (if >= 0)
                                # or disposal cost (if <= 0) of
                                # a unit of product j in period t

param cost {raw} >= 0;          # cost[i] is the storage cost
                                # per unit per period of raw material i

param value {raw};              # value[i] is the estimated residual value
                                # (if >= 0) or disposal cost (if <= 0)
                                # of raw material i after the last period

# ----------------------
# VARIABLES
# ----------------------

var Make {prd,1..T} >= 0;       # Make[j,t] is the number of units of product j
                                # manufactured in period t

var Store {raw,1..T+1} >= 0;    # Store[i,t] is the number of units of raw
                                # material i in storage at the beginning
                                # of period t

# ----------------------
# OBJECTIVE
```

```
# --------------------

maximize total_profit:
      sum {t in 1..T} ( sum {j in prd} profit[j,t] * Make[j,t] -
                        sum {i in raw} cost[i] * Store[i,t] )
    + sum {i in raw} value[i] * Store[i,T+1];

                  # Total over all periods of estimated profit,
                  # minus total over all periods of storage cost,
                  # plus value of remaining raw materials after last period

# --------------------
# CONSTRAINTS
# --------------------

subject to limit {t in 1..T}:  sum {j in prd} Make[j,t] <= max_prd;

                  # Total production in each period must not exceed maximum

subject to start {i in raw}:  Store[i,1] <= init_stock[i];

                  # Units of each raw material in storage at beginning
                  # of period 1 must not exceed initial stock

subject to balance {i in raw, t in 1..T}:

     Store[i,t+1] = Store[i,t] - sum {j in prd} units[i,j] * Make[j,t];

                  # Units of each raw material in storage
                  # at the beginning of any period t+1 must equal
                  # units in storage at the beginning of period t,
                  # less units used for production in period t

------------------------------------------------------------------------------
------------------------------------------------------------------------------
set prd := nuts bolts washers;
set raw := iron nickel;

param T := 4;
param max_prd := 123.7;

param units :     nuts    bolts   washers  :=
        iron      .79      .83      .92
        nickel    .21      .17      .08    ;

param profit :      1       2       3       4    :=
        nuts       1.73    1.8     1.6     2.2
        bolts      1.82    1.9     1.7     2.5
        washers    1.05    1.1      .95    1.33  ;

param :         init_stock    cost    value  :=
        iron       418.       .03      .02
```

```
        nickel     73.2       .025    -.01   ;
```

Et la resolution :

```
[55:~/AMPL/]%ampl
ampl: model manuf.mod; data manuf2.dat ;
ampl: solve ;
CPLEX 6.0: Tried aggregator 1 time.
Reduced LP has 6 rows, 16 columns, and 40 nonzeros.

Iteration log . . .
Iteration:     1    Objective     =             0.000000
CPLEX 6.0: optimal solution; objective 851.8190433
6 iterations (0 in phase I)
ampl: option display_1col 1 ;
ampl: display Make ;
Make [*,*] (tr)
:     bolts   nuts   washers    :=
1   123.7       0      0
2   123.7       0      0
3     5.61111   0    114.489
4   123.7       0      0
;

ampl: display Store ;
Store [*,*] (tr)
:    iron     nickel    :=
1   418       73.2
2   315.329   52.171
3   212.658   31.142
4   102.671   21.029
5     0        0
;
ampl: display start.slack, start.dual ;
:      start.slack start.dual    :=
iron         0       0.193333
nickel       0       8.54167
;

ampl: display limit.slack, limit.dual ;
: limit.slack limit.dual    :=
1      0        0.1783
2      0        0.22915
3      3.6      0
4      0        0.77085
;

ampl: display balance.slack, balance.dual ;
:         balance.slack balance.dual    :=
iron   1    0              0.223333
iron   2    0              0.253333
iron   3   -1.42109e-14    0.283333
iron   4    0              0.313333
```

```
nickel 1    0              8.56667
nickel 2    0              8.59167
nickel 3    0              8.61667
nickel 4    0              8.64167
;
ampl: quit
[56:~/AMPL/]%
```

12.2 A problème de Production.

```
####  PRODUCTION SETS AND PARAMETERS  ###

set prd 'products';    # Members of the product group

param pt 'production time' {prd} > 0;
# Crew-hours to produce 1000 units

param pc 'production cost' {prd} > 0;
# Nominal production cost per 1000, used
# to compute inventory and shortage costs


###  TIME PERIOD SETS AND PARAMETERS  ###

param first > 0 integer;
# Index of first production period to be modeled

param last > first integer;
# Index of last production period to be modeled

set time 'planning horizon' := first..last;


###  EMPLOYMENT PARAMETERS  ###

param cs 'crew size' > 0 integer;
# Workers per crew

param sl 'shift length' > 0;
# Regular-time hours per shift

param rtr 'regular time rate' > 0;
# Wage per hour for regular-time labor

param otr 'overtime rate' > rtr;
# Wage per hour for overtime labor

param iw 'initial workforce' >= 0 integer;
# Crews employed at start of first period

param dpp 'days per period' {time} > 0;
# Regular working days in a production period
```

```
param ol 'overtime limit' {time} >= 0;
# Maximum crew-hours of overtime in a period

param cmin 'crew minimum' {time} >= 0;
# Lower limit on average employment in a period

param cmax 'crew maximum' {t in time} >= cmin[t];
# Upper limit on average employment in a period

param hc 'hiring cost' {time} >= 0;
# Penalty cost of hiring a crew

param lc 'layoff cost' {time} >= 0;
# Penalty cost of laying off a crew



###  DEMAND PARAMETERS  ###

param dem 'demand' {prd,first..last+1} >= 0;
# Requirements (in 1000s)
# to be met from current production and inventory

param pro 'promoted' {prd,first..last+1} logical;
# true if product will be the subject
# of a special promotion in the period


###  INVENTORY AND SHORTAGE PARAMETERS  ###

param rir 'regular inventory ratio' >= 0;
# Proportion of non-promoted demand
# that must be in inventory the previous period

param pir 'promotional inventory ratio' >= 0;
# Proportion of promoted demand
# that must be in inventory the previous period

param life 'inventory lifetime' > 0 integer;
# Upper limit on number of periods that
# any product may sit in inventory

param cri 'inventory cost ratio' {prd} > 0;
# Inventory cost per 1000 units is
# cri times nominal production cost

param crs 'shortage cost ratio' {prd} > 0;
# Shortage cost per 1000 units is
# crs times nominal production cost

param iinv 'initial inventory' {prd} >= 0;
```

```
# Inventory at start of first period; age unknown

param iil 'initial inventory left' {p in prd, t in time}
      := iinv[p] less sum {v in first..t} dem[p,v];

# Initial inventory still available for allocation
# at end of period t

param minv 'minimum inventory' {p in prd, t in time}
      := dem[p,t+1] * (if pro[p,t+1] then pir else rir);

# Lower limit on inventory at end of period t


###  VARIABLES  ###

var Crews{first-1..last} >= 0;
# Average number of crews employed in each period

var Hire{time} >= 0;    # Crews hired from previous to current period

var Layoff{time} >= 0;  # Crews laid off from previous to current period

var Rprd 'regular production' {prd,time} >= 0;
# Production using regular-time labor, in 1000s

var Oprd 'overtime production' {prd,time} >= 0;
# Production using overtime labor, in 1000s

var Inv 'inventory' {prd,time,1..life} >= 0;
# Inv[p,t,a] is the amount of product p that is
# a periods old -- produced in period (t+1)-a --
# and still in storage at the end of period t

var Short 'shortage' {prd,time} >= 0;
# Accumulated unsatisfied demand at the end of period t



###  OBJECTIVE  ###

minimize cost:

    sum {t in time} rtr * sl * dpp[t] * cs * Crews[t] +
    sum {t in time} hc[t] * Hire[t] +
    sum {t in time} lc[t] * Layoff[t] +
    sum {t in time, p in prd} otr * cs * pt[p] * Oprd[p,t] +
    sum {t in time, p in prd, a in 1..life} cri[p] * pc[p] * Inv[p,t,a] +
    sum {t in time, p in prd} crs[p] * pc[p] * Short[p,t];

# Full regular wages for all crews employed, plus
# penalties for hiring and layoffs, plus
```

```
# wages for any overtime worked, plus
# inventory and shortage costs

# (All other production costs are assumed
# to depend on initial inventory and on demands,
# and so are not included explicitly.)


###  CONSTRAINTS  ###

rlim 'regular-time limit' {t in time}:

    sum {p in prd} pt[p] * Rprd[p,t] <= sl * dpp[t] * Crews[t];

# Hours needed to accomplish all regular-time
# production in a period must not exceed
# hours available on all shifts

olim 'overtime limit' {t in time}:

    sum {p in prd} pt[p] * Oprd[p,t] <= ol[t];

# Hours needed to accomplish all overtime
# production in a period must not exceed
# the specified overtime limit

empl0 'initial crew level':  Crews[first-1] = iw;

# Use given initial workforce

empl 'crew levels' {t in time}:  Crews[t] = Crews[t-1] + Hire[t] - Layoff[t];

# Workforce changes by hiring or layoffs

emplbnd 'crew limits' {t in time}:  cmin[t] <= Crews[t] <= cmax[t];

# Workforce must remain within specified bounds

dreq1 'first demand requirement' {p in prd}:

    Rprd[p,first] + Oprd[p,first] + Short[p,first]
     - Inv[p,first,1] = dem[p,first] less iinv[p];

dreq 'demand requirements' {p in prd, t in first+1..last}:

    Rprd[p,t] + Oprd[p,t] + Short[p,t] - Short[p,t-1]
  + sum {a in 1..life} (Inv[p,t-1,a] - Inv[p,t,a])
                          = dem[p,t] less iil[p,t-1];

# Production plus increase in shortage plus
# decrease in inventory must equal demand
```

```
ireq 'inventory requirements' {p in prd, t in time}:

    sum {a in 1..life} Inv[p,t,a] + iil[p,t] >= minv[p,t];

# Inventory in storage at end of period t
# must meet specified minimum

izero 'impossible inventories' {p in prd, v in 1..life-1, a in v+1..life}:

    Inv[p,first+v-1,a] = 0;

# In the vth period (starting from first)
# no inventory may be more than v periods old
# (initial inventories are handled separately)

ilim1 'new-inventory limits' {p in prd, t in time}:

    Inv[p,t,1] <= Rprd[p,t] + Oprd[p,t];

# New inventory cannot exceed
# production in the most recent period

ilim 'inventory limits' {p in prd, t in first+1..last, a in 2..life}:

    Inv[p,t,a] <= Inv[p,t-1,a-1];

# Inventory left from period (t+1)-p
# can only decrease as time goes on

------------------------------------------------------------------------
------------------------------------------------------------------------
set prd := _3111 _2915 _5710 _3012 _7310 _2857 _2600 _3905 ;

param first :=  1 ;
param last  := 13 ;
param life  :=  2 ;

param cs := 15 ;
param sl :=  8 ;
param iw :=  7 ;

param rtr := 16.00 ;
param otr := 55.19 ;
param rir :=  1.00 ;
param pir :=  0.80 ;


param :
pt        pc       cri       crs      iinv :=

 _3111       2.909      4775     0.015     1.100      79.2
 _2915       2.759      4520     0.015     1.100      84.7
```

```
 _5710       3.478       4600      0.015      1.100       69.6
 _3012       2.264       4520      0.015      1.100       68.0
 _7310       2.759       4150      0.015      1.100       12.9
 _2857       4.000       6720      0.015      1.100        0.0
 _2600       2.000       4150      0.015      1.100       27.9
 _3905       3.636       6410      0.015      1.100        0.0  ;


param :
    dpp         ol       cmin       cmax         hc         lc :=

  1        20.0     126.0        0.0        9.0      7500       7500
  2        19.0     126.0        0.0        9.0      7500       7500
  3        20.0     126.0        0.0        9.0      7500       7500
  4        19.0     126.0        0.0        9.0      7500       7500
  5        19.0     126.0        0.0        9.0     15000      15000
  6        19.0     126.0        0.0        9.0     15000      15000
  7        19.0     126.0        0.0        9.0     15000      15000
  8        20.0     126.0        0.0        9.0     15000      15000
  9        19.0     126.0        0.0        9.0     15000      15000
 10        20.0     126.0        0.0        9.0     15000      15000
 11        20.0     126.0        0.0        9.0      7500       7500
 12        18.0     126.0        0.0        9.0      7500       7500
 13        18.0     126.0        0.0        9.0      7500       7500  ;


param dem (tr) :

   _3111      _2915      _5710      _3012      _7310      _2857 :=

  1       102.0      143.7       67.3      109.5        7.3        0.0
  2        42.9       63.7       25.9      141.1        8.8        0.0
  3        48.3       63.7       25.9      118.1        9.4        0.0
  4       101.7      148.9       67.3      104.5        7.6        0.0
  5        42.8       63.7       20.7      169.7        8.6        0.0
  6        42.8       64.7       31.1      146.4        8.6        0.0
  7        49.3       70.5       20.7      127.4        7.1        0.0
  8        98.7      152.0       67.3      331.7        7.2        0.0
  9        43.8       70.4       25.9      133.2        8.1        0.0
 10        70.7       91.8       44.4      141.8        7.2        0.0
 11        65.6       97.1       44.4      137.4        7.9        0.0
 12        43.5       54.4       20.7      142.9        8.1        0.0
 13        49.1       74.8       25.9      382.1        8.1        0.0
 14       102.0      143.7       67.3      109.5        7.3        0.0


 :       _2600      _3905 :=

  1        29.0        0.0
  2        15.0        0.0
  3        15.0      300.0
  4        29.0        0.0
```

```
 5        15.0     200.0
 6        20.0       0.0
 7        15.0       0.0
 8        30.0       0.0
 9        15.0       0.0
10        24.0       0.0
11        24.0       0.0
12        15.0       0.0
13        20.0       0.0
14        29.0       0.0  ;


param pro (tr) :

  _3111     _2915     _5710     _3012     _7310     _2857 :=

 1           1         1         1         0         0         0
 2           0         0         0         0         0         0
 3           0         0         0         1         0         0
 4           1         1         1         0         0         0
 5           0         0         0         1         0         0
 6           0         0         0         0         0         0
 7           0         0         0         0         0         0
 8           1         1         1         1         0         0
 9           0         0         0         0         0         0
10           1         1         1         0         0         0
11           1         1         1         0         0         0
12           0         0         0         0         0         0
13           0         0         0         1         0         0
14           1         1         1         0         0         0


 :        _2600     _3905 :=

 1           1         0
 2           0         0
 3           0         1
 4           1         0
 5           0         1
 6           0         0
 7           0         0
 8           1         0
 9           0         0
10           1         0
11           1         0
12           0         0
13           0         0
14           1         0  ;
```