GUSEK –INSTALLATION ET PRISE EN MAIN

1. CONTEXTE

Les solveurs de programmation linéaire en version non limitée (CPLEX, XPRESS, LINGO, MPL etc.) sont chers, de l'ordre de 6000 euros la licence pour un PC. Excel contient un solveur mais il est limité à 300 variables et 100 contraintes et le modèle mathématique à résoudre doit respecter la logique d'Excel, avec des tables à 1 ou 2 dimensions, et donc des variables à un ou deux indices. GLPK est le premier solveur gratuit et non limité. Il s'agit d'un projet du GNU (groupement mondial des programmeurs de logiciels libres), commencé en 2000 et coordonné par Andreï Makhorin de l'Institut d'Aviation de Moscou. GLPK est une librairie de fonctions qu'on doit appeler dans des programmes en C ou Java pour définir et résoudre des programmes linéaires. Le produit est inutilisable si on n'écrit pas soi-même un programme appelant. GLPK est très fiable et peut résoudre des PL de grande taille.

Un langage de modélisation (GMPL) et un programme autonome de résolution (glpsol) ont été ajoutés ensuite. GMPL permet de taper des PL dans une syntaxe proche de l'écriture mathématique. C'est un sous-ensemble d'AMPL, un langage de modélisation très connu. On saisit le PL dans un fichier-texte (exemple "monpl.mod") avec un éditeur simple comme le bloc-notes. Pour résoudre, on ouvre une fenêtre DOS (Accessoires/Invite de commande) et on tape "glpsol –monpl.mod". L'étape suivante réalisée en 2010 a été un éditeur sous Windows, appelé GUSEK, qui permet de saisir des PL, d'appeler automatiquement le programme glpsol, et de récupérer les résultats.

2. INSTALLATION DE GUSEK POUR WINDOWS – DERNIERE VERSION 0.2.18 (26/08/2014)

Connectez-vous à l'adresse Internet suivante : http://gusek.sourceforge.net

Puis cliquer sur "Download" pour charger un fichier compressé (.zip). J'ai mis aussi ce zip sur Moodle : "gusek-a-installer". GUSEK étant une application portable, il n'y a pas de programme d'installation : il est inutile de vous connecter sur votre PC comme utilisateur "système". Il suffit de décompresser le zip dans un répertoire au choix. La seule précaution est de dire au décompresseur qu'on veut extraire les noms de chemin, car un répertoire "gusek" avec des sous-répertoires est créé.

Il est inutile d'installer le solveur de GLPK car il est inclus dans GUSEK. Pour vérifier l'installation, on peut cliquer sur l'icône de GUSEK (boule noire) : l'éditeur démarre. Vous pouvez faire un "File/Open", charger un des modèles du répertoire "Examples" comme "bpp.mod" puis faire un "Tools/Go" : le programme linéaire est alors résolu. Actuellement, il n'y a qu'une version Windows mais elle fonctionne sous Mac OS si on installe un émulateur Windows.

3. LE LANGAGE GMPL – EXEMPLE SIMPLE

Lancer GUSEK, faire un "File/New" et taper le programme linéaire suivant :

```
var x1, >= 0;
var x2, >= 0;
maximize z: 2*x1+3*x2;
subject to cont1: x1 + x2 <= 3;
subject to cont2: x2 <= 2;
solve;
display z,x1,x2;
end;
```

Le PL doit être sauvegardé une première fois pour être résolu. Créez un sous-répertoire du répertoire "gusek" pour stocker vos exemples, faites "File/Save as" avec le nom "test-1.mod" puis "Tools/Go". Un rapport d'exécution apparait dans une fenêtre à droite et donne la valeur maximale de l'objectif (8) et les valeurs des variables. Si vous modifiez le PL, les "Tools/go" suivants vont sauvegarder le fichier automatiquement. Cet exemple très simple montre déjà quelques caractéristiques de GMPL :

- toutes les instructions se terminent par un point-virgule,
- le modèle se termine par l'instruction "end",
- les mots-clés du langage ("var", "maximize", "solve", "end" etc.) sont toujours en minuscules,
- les déclarations de variables commencent par "var",
- la fonction-objectif commence par "maximize" ou "minimize",
- les contraintes peuvent commencer par "s.t." ou "subject to", mais on peut aussi ne rien mettre,
- il faut choisir un nom pour l'objectif (ici "z") et pour chaque contrainte (ici "cont1" et cont2"),
- on peut afficher les valeurs des données, des variables et de la fonction-objectif avec "display".

Vos identificateurs (variables, nom de l'objectif et des contraintes) ne doivent pas être des mots-clés. Ils peuvent contenir des lettres, des chiffres, des blancs soulignés et commencer par une lettre. Comme en C, majuscules et minuscules sont significatives : "toto" et "Toto" sont deux noms distincts. Les formules mathématiques s'écrivent comme en C et elles doivent être linéaires. Une instruction peut être tapée sur plusieurs lignes et on peut insérer des espaces ou lignes vides où on veut.

4. UN EXEMPLE PLUS COMPLEXE

GMPL permet des formules symboliques (sommes) et une séparation complète entre le modèle et les données. Retapez le PL précédent sous la forme suivante et enregistrez-le dans "test-2.mod".

```
/* Version générique du PL précédent */
param m, integer, > 0;           # Nombre de contraintes
param n, integer, > 0;           # Nombre de variables
param c {j in 1..n};             # Vecteur des coûts
param A {i in 1..m, j in 1..n};  # Matrice des coefficients des variables
param b {i in 1..m};             # Vecteur des seconds membres
var x {j in 1..n}, >= 0;         # Variables (positivité non implicite!)
maximize z: sum{j in 1..n} c[j]*x[j];
s.t. contrainte {i in 1..m} : sum{j in 1..n} A[i,j]*x[j] <= b[i];
solve;
display z,x;
data;
param m := 2;
param n := 2;
param c := 1 2 2 3;
param b := 1 3 2 2;
param A : 1 2 :=
       1  1 1
       2  0 1;
end;
```

On voit qu'on peut mettre un commentaire sur une seule ligne, après un "#". On peut aussi mettre plusieurs lignes en commentaires, entre "/*" et "*/". Dans un modèle, chaque donnée est déclarée par une instruction "param" ou "set" et chaque variable par une instruction "var".

Les déclarations servent à nommer données et variables, à préciser leur type et, pour les tableaux, à indiquer les dimensions. Les virgules sont facultatives : on peut écrire aussi "param n integer > 0;". On peut en même temps définir une valeur avec l'opérateur ":=" : "param n integer > 0 := 2". Mais il est préférable de séparer complètement le modèle des valeurs numériques, en plaçant ces dernières dans une section "data", voir plus loin.

Les données les plus simples sont des constantes, par exemple *n* le nombre de variables : "param n;". On a précisé ici que *n* est entier et strictement positif. Ces précisions servent à GUSEK pour vérifier si les données sont correctes, au moment de l'exécution. Si dans "data" on donne une valeur non entière ou négative ou nulle, on aura un message d'erreur au moment du "Tools/Go".

GMPL permet de définir des tableaux de données, comme "param c {j in 1..n};". Le nom d'indice "j" est facultatif car le"1..n" suffit pour indiquer la dimension du vecteur *c*. On peut aussi définir des matrices, comme *A*. En fait, on peut définir des tableaux jusqu'à la dimension 20 (20 indices!).

De même, on peut définir des tableaux de variables, comme *x*. On peut préciser "integer" ou "binary" pour indiquer des variables entières ou binaires. Ici, les variables sont par défaut des réels mais on a précisé qu'elles doivent être non négatives. En effet, contrairement à d'autres logiciels comme Excel, il n'y a pas de commande pour dire que les variables sont positives ou nulles par défaut.

La ligne "s.t." équivaut à "subject to". Ici, au lieu de définir deux contraintes, on a créé un groupe de contraintes qui ont la même forme, sauf que l'indice *i* varie. Cette variation de *i* s'écrit juste après le nom de la contrainte : c'est l'équivalent du "quel que soit *i*" du modèle mathématique.

En fonction de ces tableaux, on peut écrire la fonction-objectif et les contraintes avec des "sum", dans une syntaxe très proche de l'écriture mathématique. Comme en C, les indices se mettent entre crochets, sauf qu'on écrit "A[i,j]" pour les matrices au lieu de "A[i][j]" en C. Une erreur fréquente est de confondre crochets et accolades : les crochets indiquent les indices, les accolades indiquent les plages de variations de ces indices dans les "param", les "sum" et les groupes de contraintes.

La section "data" précise les valeurs des données : on reprend le mot-clé "param" et le nom de la donnée, puis on indique la valeur après ":=". Pour les tableaux, GMPL a une syntaxe un peu pénible : il faut donner chaque indice puis la valeur associée : premier indice, première valeur, deuxième indice, deuxième valeur etc. Les virgules sont facultatives. Pour *c* = (2, 3), on peut écrire au choix :

```
param c := 1 2 2 3;
param c :=
1 2
2 3;
param c := 1 2, 2 3;
```

Cette syntaxe assez lourde a un intérêt : si un tableau a beaucoup de zéros, on peut taper seulement les valeurs non nulles. Si par exemple $c_1 = 0$ et $c_2 = 3$, on peut préciser une valeur par défaut (zéro) au moment de la déclaration et indiquer seulement les valeurs non nulles dans la partie "data", ici c_2 :

```
param c default 0;  # Déclaration dans le modèle
param c := 2 3;     # Initialisation dans la partie "data"
```

On peut initialiser en parallèle plusieurs vecteurs ayant les mêmes indices. Si par exemple on a un vecteur *d* = (7,9), on peut écrire, à condition de mettre un ":" après "param" :

```
param : c d :=
1 2 7
2 3 9;
```

Les matrices comme *A* doivent aussi être initialisées en donnant les indices de chaque élément (là encore les virgules sont facultatives) :

```
param A := 1 1 1, 1 2 1, 2 1 0, 2 2 1;
```

Avec "default", on peut oublier les valeurs nulles :

```
param A default 0 := 1 1 1, 1 2 1, 2 2 1;
```

Une autre forme plus compacte et lisible est permise, en mettant les indices de colonnes entre ":" et ":=", puis de donner les lignes une par une, en commençant chaque ligne par son indice :

```
param A : 1 2 :=
      1   1 1
      2   0 1;
```

Le PL "test-2.mod" parait plus compliqué que "test-1.mod". En fait, il est facile à modifier si on ajoute des contraintes et des variables : le modèle, qui est paramétré par *m* et *n*, reste le même, seule la partie "data" doit être modifiée.

5. EXTENSIONS

GMPL offre des mécanismes pour définir des ensembles et des indexations complexes. Par exemple, si on a *n* produits, on peut définir un ensemble d'indices "produits", qui peut être utilisé pour indexer des tableaux et des sommes :

```
param n;
set produits := {1..n};       # Ou : 1..n car l'opérateur .. renvoie un ensemble
param cout {i in produits};   # Ou : param couts {produits};
...
s.t. contrainte : sum {i in produits} x[i] <= 3;
```

On peut aussi mettre des conditions sur les indices, après un ":". Par exemple, l'instruction suivante indique une somme des variables x_i telles que $prix_i \geq 200$, sur l'ensemble des indices de produits :

```
sum {i in produits : prix[i] >= 200} x[i]
```

Ces conditions peuvent concerner les indices et les données, mais pas les variables. Si vous voulez exprimer une condition sur des variables, elle doit être une des contraintes du modèle.

On peut faire des ensembles de couples. L'exemple suivant définit un ensemble d'étudiants E indicé de 1 à n, un ensemble de binômes B et un tableau des notes obtenues par les binômes ($note$).

```
param n integer;                 # Nombre d'étudiants
set E := 1..n;                   # Ensemble des étudiants
set B within E cross E;          # Ensemble de binômes B inclus dans E x E
param note {(i,j) in B};         # Ensemble de notes indicées par les binômes
...

data;
param n := 10;
set B :=
1 2
6 5
3 9
8 4
6 7;
param note :=
1 2 14
6 5 14
3 9 18
8 4 15
6 7 08;
end;
```

L''ensemble E des indices d'étudiants est déclaré comme l'intervalle des entiers de 1 à *n* et celui des binômes est défini comme un sous-ensemble du produit cartésien *E* x *E*, noté "E cross E". "within" désigne l'inclusion (⊆) tandis que "in" signifie l'appartenance d'un élément à un ensemble (∈).

Notez que ces déclarations ne définissent pas les valeurs (les contenus) de *E* et *B* car *n* est inconnu pour l'instant : les valeurs sont précisées ici dans la section "data". Pour B, on donne simplement la liste de ses couples et GUSEK va vérifier que les éléments de ces couples sont bien dans E. On voit qu'il est facile de définir des tableaux de données indicées par des couples, ici le tableau $note$. Avec la même syntaxe (sauf le mot var au lieu de $param$) on peut définir des tableaux de variables.

Dans l'exemple précédent, on précise deux fois les couples : dans l'initialisation de B et celle de $note$. On peut omettre le "set B := …" et initialiser B et $note$ en parallèle (noter les deux points) :

```
param : B : note :=
1 2 14
6 5 14
3 9 18
8 4 15
6 7 08;
```

Dans les équations d'un PL, il faudrait théoriquement écrire $note[(i,j)]$ pour la note du binôme (i,j). Pour simplifier, GMPL utilise la même notation que pour une matrice, c'est-à-dire $note[i,j]$.

Quel est l'intérêt de ces déclarations? On aurait pu définir $note$ comme une matrice $n \times n$ indicée par les deux étudiants de chaque binôme possible, mais pour 10 étudiants cette matrice a 100 éléments. Dans notre exemple, les 10 étudiants sont déjà groupés en 5 binômes et le tableau $note$, défini seulement pour ces binômes, a 5 éléments. On consomme donc moins de mémoire.

Si dans le modèle une somme fait référence au binôme (1,3), qui n'existe pas, on aura une erreur au moment de la résolution. Pour calculer par exemple la somme des notes sans connaître la liste exacte des binômes, il suffit d'écrire :

```
param somme := sum{(i,j) in B} note[i,j];
```

En effet, GUSEK va lire la section $data$ avant de résoudre le modèle : il saura donc très bien quelle est la liste des binômes existants et fera la somme correctement!

Le manuel "gmpl.pdf" fourni avec GUSEK décrit toutes les fonctionnalités mais il donne la syntaxe avec très peu d'exemples. Je vous conseille plutôt de regarder les modèles du répertoire "Examples" de GUSEK, qui sont bien commentés et contiennent des instructions très variées.

6. ORDRE DES ACTIONS DANS GUSEK

Même s'il n'y a pas encore de section "data", on peut déjà faire des "Tools/Go" pour vérifier la syntaxe. En fait, l'ordre des actions est le suivant quand on fait un "Tools/Go" :

- analyse de la syntaxe et signalement des erreurs éventuelles (mot-clé inconnu etc.),
- lecture des données de la section "data" avec vérification des types (integer etc.) et des indexations,
- génération du PL numérique (tableau du simplexe) dans un fichier et appel du solveur GLPK,
- récupération des résultats de GLPK et affichages.

Attention! Dans les messages d'erreur, GUSEK indique la ligne où il ne comprend plus rien, mais l'erreur vient souvent d'une ligne au-dessus (point-virgule ou accolade manquante par exemple).

GUSEK – UTILISATION AVANCEE

1. STRUCTURE D'UN FICHIER-MODELE

Les fichiers-modèles sont des fichiers-texte avec l'extension ".mod" et la structure générale suivante :

```
déclaration des données et variables
définition de l'objectif et des contraintes
solve;
écriture éventuelle des résultats
data;
initialisation des données (:=)
end;
```

Il n'y a pas d'instruction spéciale pour le début de modèle mais il faut finir par "end". Les déclarations définissent des ensembles avec l'instruction "set", des scalaires ou des tableaux avec "param" et des variables avec "var". L'objectif commence par "minimize" ou maximize", suivi d'un nom et d'un ":". Les contraintes débutent par "subject to" ou "s.t." (facultatifs), suivi du nom de la contrainte et ":".

L'instruction "solve" n'est nécessaire que si on veut écrire des résultats, qui sont connus seulement quand le modèle a été résolu. On peut initialiser certaines données au moment de leur déclaration, mais il est conseillé de le faire dans une section "data", pour bien séparer le modèle des valeurs particulières considérées pour une résolution. Cette section "data" peut être placée dans un fichier de même nom que le modèle mais avec l'extension ".dat" (il faut copier le "end"). Si on coche "Tools/Use external .dat", GUSEK va rechercher le fichier ".dat" associé au modèle. On peut ainsi définir plusieurs jeux de données (instances) sans avoir à changer le fichier-modèle.

Les mots-clés du langage GMPL comme "data" sont toujours en minuscules. Vos identificateurs peuvent contenir des lettres de "a" à "z", des chiffres et des blancs soulignés ("_"). Comme en C, le langage distingue majuscules et minuscules : "Toto" et "toto" sont deux identificateurs différents.

Toutes les instructions sont terminées par des points-virgules. On peut insérer des espaces partout et écrire une instruction sur plusieurs lignes. De nombreuses instructions prévoient des virgules, mais elles sont optionnelles. On peut mettre des commentaires partout, après un "#" (commentaire limité à une ligne) ou entre "/*" et "*/" (dans ce cas le commentaire peut s'étaler sur plusieurs lignes).

2. DECLARATION DE SCALAIRES

Une donnée scalaire (c'est-à-dire une valeur unique, contrairement à un tableau ou un ensemble) se déclare avec "param". On peut préciser un type : "integer", "binary" (0 ou 1) ou "symbolic" (chaîne de caractères entre deux guillemets ou apostrophes), et une condition précisant les valeurs autorisées. Par défaut, la donnée est considérée comme un nombre réel. Les virgules sont facultatives. Il est important de préciser quelles données sont entières, car les calculs sont plus rapides et ont une précision absolue. Les conditions sur les valeurs sont utilisées par GUSEK pour vérifier les données au moment de la résolution : elles vous protègent des erreurs de saisie.

```
param n;
param u, integer;
param v, integer, >= 0, <= 10;
param w integer >= 0;
param q integer in {2,7,9};
param region symbolic;
```

On peut déclarer et initialiser en même temps une donnée scalaire, avec l'opérateur ":=". La valeur d'initialisation peut être une constante ou une formule combinant des données déclarées avant. Contrairement aux contraintes du modèle, qui doivent être linéaires, les formules d'initialisation peuvent être non linéaires : ci-dessous m est initialisé avec la partie entière du sinus de n (voir les fonctions disponibles page 14 du manuel "gmpl.pdf").

```
param n integer >= 0 := 10;
param m integer := floor (sin(n));
param region symbolic := "alsace";
param prenom symbolic := 'jean-pierre';
```

Les initialisations sont réalisées seulement au moment de la résolution. Dans l'exemple précédent, l'initialisation de *m* est correcte même si *n* n'est pas initialisé maintenant : il faudra cependant que *n* soit initialisé dans la section "data" sinon *n* et *m* seront indéfinis au moment de la résolution.

La syntaxe pour une variable scalaire est similaire, à part le mot-clé "var". Mais les variables sont toujours numériques (pas de type "symbolic") et on ne peut pas leur affecter une valeur avec ":=". Les variables déclarées sont incluses dans le PL seulement si elles sont réellement utilisées, dans la fonction-objectif et/ou les contraintes. Si aucun type n'est précisé, on obtient un PL à variables continues. Si au moins une variable a le type "integer" ou "binary", on a un PL mixte.

```
var x >= 0;
var y integer >= 0 <= 30;
var z binary;
var w = 3.5;
```

Attention! Si une variable doit être non négative, il faut le dire, même pour des variables "integer" : contrairement à des logiciels comme Excel, il n'y a pas de commande pour dire que les variables sont non négatives par défaut. Un oubli donne souvent une erreur du genre "pas d'optimum fini" à la résolution, en minimisation : en effet, le solveur peut mettre les variables à moins l'infini pour minimiser le coût! Les conditions sur les valeurs ne peuvent être que des bornes inférieures et supérieures par rapport à des constantes. Elles sont considérées comme des contraintes du PL. Le signe = dans l'exemple ci-dessous est en fait une contrainte d'égalité.

3. DECLARATION D'ENSEMBLES

En GMPL, un ensemble de nombres ou de chaînes de caractères se déclare avec "set". Comme dans "param", des conditions peuvent s'appliquer aux éléments. La plus courante est de restreindre un ensemble pour qu'il soit inclus dans un autre, avec "within". La troisième ligne ci-dessous définit un sous-ensemble de couples de l'ensemble "noms", "cross" désignant le produit cartésien.

```
set noms;
set quelques_noms within noms;
set binomes within noms cross noms;
```

Pour initialiser un ensemble au moment de sa déclaration, on utilise l'opérateur d'affectation ":=" suivi de la liste des valeurs des éléments entre deux accolades (les virgules sont optionnelles) :

```
set ensemble := {1, 7, 8};
set noms := {"dupont", "durand", "martin","dupond"};
set binomes within noms cross noms := {("dupont","durand"), ("dupont","martin")};
```

Comme pour les paramètres scalaires, on peut initialiser un ensemble avec des formules. Elles peuvent combiner des calculs numériques et des opérations d'ensembles, voir pages 22-25 du manuel gmpl.pdf. L'opérateur ensembliste le plus simple est l'opérateur de génération d'intervalle "..", autorisé pour des entiers. Comme le résultat est un ensemble, les accolades sont facultatives.

Les exemples suivants utilisent les opérateurs ".." (construction d'un intervalle), "union", "by" (intervalle contenant un entier sur deux), "cross" (génération du produit cartésien) et "setof" (construction d'ensemble basée sur une itération). Dans la déclaration de $mois$, on peut omettre les accolades car le résultat de l'opérateur ".." est déjà considéré comme un ensemble.

```
set mois := {1..12};                    # Ou : set mois = 1..12;
param n integer > 0;
set intervalle := 1..n;                 # Intervalle paramétré avec n
set liste := 1..10 union {13,15};       # Union des entiers de 1 à 10 plus 13 et 15
set impairs := 1..11 by 2;              # Contient 1, 3, 5, 7, 9, 11
set couples := liste cross liste;       # Contient tous les couples de liste × liste
set doubles := setof {i in liste} 2*i; # Eléments de liste, multipliés par 2
```

4. DECLARATIONS DE TABLEAUX

Les tableaux de scalaires (numériques ou symboliques) ou de variables ont un domaine d'indices entre accolades. De tels domaines sont aussi utilisés pour les "sum" ou pour définir des groupes de contraintes. On peut utiliser jusqu'à 20 indices. Ils ont n'importe quel nom, comme en maths.

Chaque indice appartient à un intervalle d'entiers comme 1..*n* ou à un ensemble. On peut préciser un type, une condition ou une valeur par défaut, mais ils s'appliquent à tous les éléments du tableau.

```
param n integer >= 0;
set mois := {"Jan", "Fev", "Mar", "Avr"};
param b {i in 1..n} integer default 0;
param C {i in 1..n, j in 1..n} integer >= 0 default 0;
param duree {i in mois};
var x {in in 1..n, j in 1..n} binary;
```

On peut indicer par des couples, ce qui est très utile pour les graphes. Dans l'exemple précédent, la matrice *C* peut désigner des coûts sur les arcs (*i*, *j*) d'un graphe. On stocke n^2 éléments même si le graphe a peu d'arcs. Si on déclare un ensemble de nœuds *V* et un ensemble d'arcs *A* (sous-ensemble des couples possibles de *V*), on peut redéfinir *C* comme un tableau de coûts indicés par *A* :

```
set V := 1..n;              # Ensemble des noeuds du graphe
set A within V cross V;     # Ensemble des arcs (couples de V x V)
param C {(i,j) in A}, > 0;  # Capacités sur les arcs
```

Les éléments de *C* sont notés *C*[*i,j*] et non *C*[(*i,j*)], comme pour une matrice. L'avantage est de stocker uniquement les coûts des arcs existants : si *A* = {(1,3), (4,2)}, seuls *C*[1,3] et *C*[4,2] vont exister.

Un tableau à indices entiers peut avoir des "trous", comme "ventes" ci-dessous :

```
set mois_31_jours := {1,3,5,8,10,12};
param ventes {i in mois_31_jours};
```

Les domaines peuvent préciser une condition analogue au "tel que" en maths, après un ":". Par exemple, le tableau suivant a pour indices les entiers multiples de 3 entre 1 et 15 :

```
param T {i in 1..15 : i mod 3 = 0);
```

Attention! Contrairement aux ensembles, on peut initialiser un tableau par une liste de valeurs uniquement dans une section "data". L'opérateur ":=" dans une déclaration permet seulement de préciser une seule valeur ou une expression de calcul pour qui est affectée à chaque élément du tableau. Ainsi, la ligne 3 suivante permet de définir une matrice identité *C* grâce à "if", tandis que *h* est initialisé avec les parties entières (*floor*) des sinus des entiers de 1 à *n*.

```
param b {i in 1..n} := 1;
param C {i in 1..n, j in 1..n} := i+j;
param C {i in 1..n, j in 1..n} binary := if i = j then 1 else 0;
param h {i in 1..n} integer := floor(sin(i));
```

A noter : dans le cas de la matrice binaire *C*, "binary" représente les entiers 0 et 1, tandis qu'une comparaison vaut *true* ou *false*. Il est donc impossible d'écrire la ligne suivante, il faut un "if" :

```
param C {i in 1..n, j in 1..n} binary := i = j;
```

Le remplissage d'un tableau grâce à une formule de calcul est une fonctionnalité puissante. Ainsi, on peut calculer une matrice de distances euclidiennes D pour n points du plan définis par leurs coordonnées X_i et Y_i (n et les coordonnées seront initialisés dans la partie "data"). On utilise la formule de la distance euclidienne, avec racine carrée $sqrt$ et l'exponentiation "^" ou "**" :

```
param n integer > 0;
param X {i in 1..n};
param Y {i in 1..n};
param D {i in 1..n, j in 1..n} := sqrt ((X[i]-X[j])^2 + (Y[i]-Y[j])^2);
```

Comme dans des contraintes, on peut également utiliser des sommes. Par exemple, dans le tableau T suivant, T_i donne la somme des entiers de 1 à i :

```
param T {i in 1..n} := sum {j in 1..i} j;
```

Dans ces deux derniers exemples, on voit qu'on n'écrit pas une boucle pour initialiser les tableaux : on donne une formule de calcul en fonction des indices, et GUSEK fait automatiquement varier les indices pour remplir le tableau.

Le manuel gmpl.pdf liste les opérateurs dits "itérés" comme "sum" (p. 15), les opérateurs arithmétiques disponibles comme "mod" (p. 16), les fonctions mathématiques comme "sin" (p. 14), les opérateurs relationnels comme ">=" (p. 26), les opérateurs logiques comme "and" (p. 27).

En particulier, "sum" n'est pas le seul opérateur itéré : GMPL propose aussi "prod" (Π en maths), "min" et "max". On peut les utiliser dans des expressions portant sur des données et des indices, mais pas sur des variables car sinon on obtient un modèle non linéaire. Exemple :

```
param T {i in 1..n};
param Tmax := max {i in 1..n} T[i];
```

Signalons pour terminer cette section qu'on peut définir des tableaux d'ensembles, en utilisant le mot-clé "set" au lieu de "param". Ainsi, au lieu de l'ensemble d'arcs A de la page 8, on peut définir un ensemble V de nœuds puis, pour chaque nœud i, un sous-ensemble de nœuds-successeurs $S[i]$:

```
param n integer;
set V := 1..n;
set S {i in V} within V default {};
```

Comme pour les tableau simples, S doit être initialisé dans la section "data", voir la section 5.4.

5. SECTION "DATA"

5.1 Scalaires

Les initialisations peuvent être placées dans une section "data" pour séparer le modèle des données. Les initialisations avec des formules (cas de *m*) ne doivent pas être mises dans "data" car elles se déduisent d'autres données. Pour *p*, GUSEK signale une erreur car la valeur affectée est négative.

```
param n integer >= 0;
param m integer := 2*n;
param p >= 0;
param region symbolic;
…
data;
param n := 10;
param p := -1;
param region := "alsace";
end;
```

Pour les paramètres symboliques, les guillemets dans "data" sont optionnels si les chaînes contiennent seulement des lettres non accentuées, des chiffres, des blancs soulignés ou des tirets :

```
param region := alsace;
param region := haute-normandie;
```

Les lignes suivantes sont fausses à cause des espaces ou accents, il faut mettre des guillemets :

```
param region := midi-pyrénées;
param region := champagne ardenne;
```

5.2 Ensembles

Pour un ensemble, la section "data" doit donner une liste d'éléments, avec ou sans virgules, mais sans accolades. Ci-dessous, "noms" est égal à l'ensemble vide ("default {}") si on l'oublie dans "data". Comme pour les paramètres, les initialisations avec des formules de calculs ne peuvent pas être mises dans "data" : c'est le cas de l'ensemble "tous_les_binomes" qui est calculé comme le produit cartésien noms x noms, et de "mois" car ".." est considéré comme un opérateur de calcul.

```
set ensemble;
set mois := 1..12;
set noms default {};
set binomes within noms cross noms;
set tous_les_binomes := noms cross noms;
…
data;
set ensemble := 1, 7, 8;
set noms := "dupont", "durand", "martin", "dupond";
set binomes:= (dupont,durand) (dupont,martin);
```

5.3 Tableaux de scalaires

Contrairement aux données scalaires et aux ensembles, la section "data" est obligatoire pour affecter une valeur spécifique à chaque élément d'un tableau de données. On doit donner une liste entre virgules, avec l'indice et la valeur de chaque élément. Les virgules sont facultatives. On peut mettre les indices entre crochets pour améliorer la lisibilité. Par exemple, si *b* a été déclaré comme un tableau d'entiers indicé de 1 à 3, on peut écrire au choix pour *b* = (12, 53, 17) :

```
param b := 1, 12, 2, 53, 3, 17;
param b := 1 12 2 53 3 17;
param b :=
1 12
2 53
3 17;
param b := [1] 12, [2] 53, [3] 17;
```

Pour un tableau "jours" indicé sur un ensemble de mois {"Jan", "Fev", "Mar", "Avr"}, on doit aussi préciser les indices. Crochets et virgules sont facultatifs. Comme dans le cas de *region* vu avant, les guillemets sont aussi optionnels car les noms sont des identificateurs GMPL valides :

```
param jours := ["Jan"] 31, ["Fev"] 28, ["Mar"] 31, ["Avr"] 30;
param jours := Jan 31 Fev 28 Mar 31 Avr 30;
```

Si un tableau a beaucoup de zéros, on peut taper seulement les valeurs non nulles à condition de préciser 0 par défaut dans la déclaration. Si par exemple *b* est nul sauf *b*[2] = 53, on peut écrire :

```
param b {i in 1..n} default 0;
…
data;
param b := 2 53;
```

Reprenons l'exemple du graphe de la section 4 page 8, avec un ensemble d'arcs *A* et un tableau de coûts *C* indicés par des arcs. On peut initialiser *A* puis *C* comme ci-dessous, dans la section "data" :

```
set A := (1,3) (4,2);
param C := [1,3] 17, [4,2] 95;
```

Pour les matrices, il faut donner les deux indices. Par exemple, pour une matrice *C*, 2 x 2 :

```
param C := [1,1] 1, [1,2] 1, [2,1] 0, [2,2] 1;
```

Ou sans crochets ni virgules (mais peu lisible) :

```
param C := 1 1 1 1 2 1 2 1 0 2 2 1;
```

Si on a mis "default 0" dans la déclaration, on peut donner seulement les valeurs non nulles :

```
param A := [1,1] 1, [1,2] 1, [2,2] 1;
```

Un autre format plus compact est autorisé pour les matrices. On rappelle les indices des colonnes entre ":" et ":=", puis on donne les lignes une par une, en commençant par l'indice de ligne :

```
param C : 1 2 :=
      1   1 1
      2   0 1;
```

Ce format est aussi valable si les indices sont symboliques, par exemple des noms de mois.

5.4 Tableau d'ensembles

Reprenons l'exemple du tableau d'ensembles de la page 9, pour un graphe défini par un ensemble de nœuds V et un tableau donnant l'ensemble S_i des nœuds-successeurs de chaque nœud i.

```
param n integer;
set V := 1..n;
set S {i in V} within V default {};
…
data;
param n := 5;
set S[1] := 3 4;
set S[3] := 1 4 5;
```

La seule contrainte pour l'initialisation est de définir un par un chaque sous-ensemble. En effet, comme les sous-ensembles peuvent avoir des tailles différentes, GUSEK ne pourrait pas savoir où s'arrête la liste des éléments d'un sous-ensemble si on initialisait tout dans la même instruction. Dans cet exemple, notez qu'on peut ne pas donner les sous-ensembles $S[2]$, $S[4]$ et $S[5]$ car par défaut les éléments du tableau S sont initialisés avec l'ensemble vide.

5.5 Initialisation de tableaux en parallèle

La section "data" permet d'initialiser en parallèle plusieurs tableaux à une dimension ayant les mêmes indices, comme s'ils formaient une table (*tabbing format*). Si on a deux tableaux indicés de 1 à 3, *b* = (12,53,17) et *g* = (67,59,22), on peut écrire, à condition de mettre un ":" après "param" :

```
param : b, g :=
1 12 67
2 53 59
3 17 22;
```

Les virgules dans la liste de tableaux sont facultatives. On peut initialiser des vecteurs n'ayant pas les mêmes indices, en indiquant les valeurs inexistantes par des points. Si *g* est indicé de 2 à 4 :

```
param : b g :=
1 12  .
2 53 67
3 17 59
4  . 22;
```

Le "tabbing format" s'applique aussi à des tableaux indicés sur un ensemble :

```
set mois;
param ventes {i in mois};
param profit {i in mois};
…
data;
set mois := Jan, Fev, Mar;
param : ventes, profit :=
Jan 33 5
Fev 49 7
Mar 66 6;
```

On a initialisé l'ensemble "mois" avant d'initialiser en parallèle "ventes" et "profit". Le "tabbing format" permet aussi de faire les 3 initialisations si on met un ":" entre l'ensemble et les tableaux :

```
param : mois : ventes, profit :=
Jan 33 5
Fev 49 7
Mar 66 6;
```

Pour des tableaux indicés par des couples, le "tabbing format" doit préciser les couples. Par exemple, si *A* est l'ensemble des arcs d'un graphe, *Capa* un tableau de capacités sur les arcs et *Cost* un tableau de coûts sur les arcs :

```
set A := (1,3) (4,2) (7,1);
param : Capa Cost :=
1 3 19 24
4 2 41 35
7 1 22 57;
```

Comme dans l'exemple sur les ventes et les profits, on peut cependant initialiser *A*, *Capa* et *Cost* en même temps (on enlève donc la ligne avec "set A := …") :

```
param : A : Capa Cost :=
1 3 19 24
4 2 41 35
7 1 22 57;
```

Il n'est pas possible de mettre deux tableaux rectangulaires côte à côte dans le" tabbing" format, mais on peut les initialiser en parallèle en donnant la liste des couples d'indices. Ci-dessous, la colonne 3 du dernier "param" donne les éléments de T et la colonne 4 ceux de H :

```
param T {i in 1..2, j in 1..2};
param H {i in 1..2, j in 1..2};
…
data;
param : T H :=
1 1 10 23
1 2 15 27
2 1 29 56
2 2 78 76;
```

6. FICHIERS DATA

On peut enlever la partie "data" d'un modèle et la mettre dans un fichier séparé (on garde le "end" dans le modèle). Cette pratique est conseillée pour les gros modèles à données changeant souvent. En effet, on utilise souvent une application (programme VBA, ERP, base de données) pour préparer le fichier de données. Si votre fichier-modèle s'appelle "nom.mod", le fichier de données doit s'appeler "nom.dat", commencer par "data" et finir aussi par "end". Si vous cochez "Tools/Use external .dat", GUSEK va chercher le fichier ".dat" associé à chaque résolution avec "Tools/Go". Vous aurez une erreur si le modèle contient déjà une section "data" ou si le fichier-données n'est pas trouvé.

On peut même avoir plusieurs fichiers de données pour le même modèle, et avec des noms quelconques. Supposons que "toto.mod" ait deux fichiers de données "test-1.dat" et "test-2.dat".

Pour les utiliser, a) ouvrez les trois fichiers dans GUSEK et cochez "Tools/Use external .dat file" comme précédemment; b) mettre la fenêtre active sur le fichier de données qui vous intéresse, par exemple "test-1.dat" et faire "Tools/Set as default .dat file"; c) mettre la fenêtre active sur le modèle et faire un "Tools/Go" (si on reste sur la fenêtre des données, on a une erreur car GUSEK attend un modèle). Pour résoudre un autre jeu d'essai, refaire les étapes b) et c) avec le nouveau fichier.

7. AFFICHAGES PAR DISPLAY ET PRINT

L'inconvénient de GUSEK par rapport au solveur d'Excel est que seule la valeur optimale de l'objectif est affichée à l'écran en fin d'exécution : on ne voit ni les quantités calculées (combinaisons linéaires des contraintes) ni les valeurs des variables. La solution détaillée peut être obtenue dans un fichier ".out" si on coche "Tools/Generate output file on Go", mais on obtient un listing très technique et difficile à comprendre pour des non-spécialistes.

On peut écrire soi-même des données, des variables, la valeur de l'objectif, des valeurs de contraintes (valeur de la combinaison linéaire de la contrainte) et même des quantités calculées quelconques dans la fenêtre d'exécution de GUSEK (à droite de celle du modèle) grâce aux instructions "display" et "printf". Ces instructions peuvent être placées n'importe où dans le modèle, sauf dans la partie "data". Comme GUSEK initialise les données avant de faire les écritures, le display suivant affiche correctement "n = 3" :

```
param n;
display n;
…
data;
param n := 3;
```

Si on veut écrire des valeurs qui dépendent des variables, il faut d'abord résoudre le modèle avec l'instruction "solve". Cette instruction est inutile si on veut afficher des données ou des valeurs calculées uniquement à partir des données :

```
var x;
…
solve;
display x;
```

L'instruction "display" est rudimentaire mais facile à utiliser. Elle sert souvent pour vérifier le contenu d'un tableau. Elle précise une liste d'items à afficher. GUSEK écrit un message en rouge avec le n° de ligne du "display" avant de l'exécuter, ce qui permet de distinguer entre plusieurs "display" du même tableau. Puis il écrit chaque item sur une ligne, avec son nom. Ainsi, si le modèle contient :

```
param T {i in 1..2,j in 1..2};
display T; # Supposons qu'on soit à la ligne 19
…
data T : 1 2 :=
1 10 15
2 29 78;
```

On obtient les lignes suivantes dans la fenêtre d'exécution, avec les indices du tableau :

```
Display statement at line 19
T[1,1] = 10
T[1,2] = 15
T[2,1] = 29
T[2,2] = 78;
```

On peut spécifier un domaine. Par exemple, pour afficher la diagonale de T :

```
display {i in 1..2, j in 1..2 : j = i} T[i,j];
```

Ce qui donne à l'exécution :

```
Display statement at line 196
T[1,1] = 10
T[2,2] = 78
```

Voici trois exemples plus complexes avec le PL générique de la page 2 :

```
display z, x{j in 1..n}, contrainte{i in 1..m};
display z, x, contrainte;                       # Version sans les indices
display contrainte[2];                          # Combinaison linéaire contrainte 2
display "Somme des x[i]:", sum{j in 1..n} x[j]; # Exemple d'expression calculée
```

Dans le 3ème exemple, notez qu'on met des crochets pour faire référence à la deuxième contrainte. Pour le quatrième exemple, on aurait pu définir une variable "sx" et une contrainte pour la calculer :

```
var sx;
calcul_sx : sx = sum {j in 1..n} x[j];
display sx;
```

La grosse différence est qu'on augmente la taille du modèle et le temps de résolution!

L'instruction "printf" permet des formatages et elle n'est pas accompagnée d'un message en rouge à l'exécution. Elle est similaire au "printf" du langage C sauf que les arguments ne sont pas entre parenthèses et qu'on peut préciser un domaine comme pour les "display". Sa syntaxe est la suivante :

```
printf {domaine facultatif} "formats", liste d'items séparés par des virgules;
```

La chaîne de format peut contenir du texte et des spécifications de formats commençant par un "%". Les items successifs sont écrits en suivant les formats successifs indiqués. Par exemple, "%d" affiche un entier. Si x = 541 et y = -3, l'instruction suivante :

```
printf "x = %d, y = %d\n", x, y;
```

Affiche le texte ci-dessous :

```
x = 541, y = -3
```

Le "\n" indique qu'un saut de ligne doit être effectué à la fin du "printf". Si on ne le met pas, le prochain "printf" continue sur la même ligne. Voici un exemple simple avec un domaine :

```
printf {1..20} "-"; printf "\n";   # Affiche une 20 tirets puis va à la ligne;
```

Les principaux formats sont les suivants :

- %d pour un entier. On peut imposer le nombre de positions à écrire, par exemple %5d.
- %s pour une chaîne de caractères. Là aussi, on peut préciser le nombre de positions : %15s.
- %f pour un réel : %5.2f spécifie 5 positions en tout dont 2 décimales, %.2f va prendre autant de positions que nécessaire pour la partie entière, mais toujours avec 2 décimales.

Si le nombre de positions est plus grand que le contenu à afficher, ce dernier est justifié à droite, ce qui est inhabituel pour des chaînes de caractères. Par exemple, un format %5s pour "Oui" donne " Oui" avec deux blancs. On peut cadrer à gauche en mettant un tiret après le %, par exemple %-5s.

Une instruction "for" permet de répéter un bloc d'instructions entre accolades, en précisant un domaine. Une boucle "for" ne peut contenir que des "display", des "printf" et des autres "for". Par exemple, pour afficher la matrice T de la page précédente proprement, avec 2 positions par élément, un espace de séparation et les indices, on peut écrire :

```
printf " ";
for {j in 1..2} printf " %2d",j;   # Ou : printf {j in 1..2} " %2d",j;
printf "\n"
for {i in 1..2} {
   printf "%d",j;
   for {j in 1..2} printf " %2d", T[i,j];
   printf "\n"
}
```

On obtient :

```
   1  2
1 10 15
2 29 78
```

Les accolades peuvent être omises après le premier et le troisième "for" car ils ne contiennent qu'une instruction. Ce genre de "for" peut être remplacé par un "printf" avec un domaine :

```
printf {j in 1..2} " %2d", T[i,j]
```

Le "for" de GUSEK est équivalent à celui du C mais il peut faire varier plusieurs indices. Par exemple, on peut écrire les éléments de la matrice T (un sur chaque ligne) comme suit :

```
for {i in 1..2, j in 1..2} printf "T(%d,%d) = %d\n", i, j, T[i,j];
```

Autre exemple : un tableau donne le nombre de touristes ayant visité trois villes au cours des trois premiers mois de l'année. On veut écrire ce tableau en indiquant pour chaque ligne la ville, cadrée à gauche sur 9 positions, suivie des nombres de touristes sur 4 positions, séparés par des virgules.

```
set villes := {"Paris","Marseille","Aix"};
set mois := {1..3};
param touristes {i in villes, j in mois};
for {i in villes} {
   printf "%-9s:", i;
   for {j in mois} printf "%4d,", touristes[i,j];
  printf "\n";
}
data;
param touristes : 1   2   3 :=
Paris            18 175 132
Marseille       901 430  17
Aix              45  12 606;
```

On obtient l'affichage suivant. Si on avait mis "%s" pour le premier "printf", les ":" n'auraient pas été alignés verticalement. Avec "%9s", les noms de villes auraient été cadrés à droite.

```
Paris    :  18, 175, 132,
Marseille: 901, 430,  17,
Aix      :  45,  12, 606,
```

On a souvent besoin de conserver les résultats affichés. Une première façon est de sélectionner avec la souris ce qui nous intéresse, dans la fenêtre d'exécution de GUSEK, et de copier dans un autre logiciel. Une autre manière est d'écrire sur un fichier-texte. En C, il faut utiliser une instruction "fprintf". En GMPL, on utilise "printf" avec le nom du fichier après ">" ou ">>". Le ">" indique que le fichier est effacé s'il existe déjà, avant d'écrire. Le ">>" écrit après le contenu actuel du fichier :

```
printf "Coucou!\n" > "resultats.txt";
printf "C'est moi!\n" >> "resultats.txt";
```

Le ">" est utile pour effacer le fichier précédent quand on fait plusieurs exécutions. Le nom de fichier peut être mis dans une chaîne de caractères (paramètre "symbolic"), ce qui permet de faire des rapports sur des fichiers différents à chaque exécution, sans avoir à changer tous les "printf" :

```
param fichier symbolic := "resultats.txt";
printf "Coucou!\n" > fichier;
printf "C'est moi!\n" >> fichier;
```

Le fichier-texte obtenu peut ensuite être ouvert avec GUSEK, le bloc-notes, Word ou Excel.

Attention! Le petit langage ressemblant au C dans GMPL ne sert qu'à faire des écritures, dans la version actuelle. Il ne permet pas de faire des tests ("if"), de déclarer des variables locales ou de calculer des expressions comme "i = 2*i +j;". Les logiciels de modélisation OPL-STUDIO et XPRESS, plus complexes, offrent un langage de programmation complet pour faire ce qu'on veut.

8. TABLES

8.1 Format CSV

Au lieu d'initialiser un tableau (ou plusieurs tableaux en parallèle) dans la section "data", on peut mettre les valeurs dans un fichier Excel au format CSV et lire ce fichier avec une instruction "table". Cette instruction est autorisée entre les déclarations des tableaux concernés et la fonction-objectif du modèle. Comme "printf" et "display", elle est interdite dans la section "data". Cette technique est utile pour éviter de mettre des gros tableaux (comme des distanciers) directement dans les modèles.

Le format CSV est un type simple de fichier Excel, contenant uniquement une plage rectangulaire de cellules avec des nombres ou du texte sans espaces (pas de formules, de couleurs, de cadres etc.). Ce fichier est codé avec du texte normal (pas de gras, souligné etc.) et peut être tapé et relu avec un éditeur de texte simple comme le bloc-notes ou l'éditeur de GUSEK. Dans le fichier, les contenus des cellules sont séparés par des virgules. Chaque ligne se termine avec un saut de ligne (LF) et non pas par un retour-chariot + saut de ligne (CR+LF) comme dans les fichiers-texte de DOS ou Windows.

Avec un Excel US, on peut taper un fichier CSV et faire "Save as" en choisissant le format CSV : le fichier aura un format correct. Par exemple, tapez avec un Excel US la feuille suivante et enregistrez-la sous le nom "myfile", en choisissant comme format "CSV", ce qui donne un fichier "myfile.csv" :

Ventes	10.7	12
Profits	2	4
Achats	8	9

Si on ouvre le fichier dans GUSEK, on voit le texte suivant (les bordures n'ont pas été gardées) :

```
Ventes,10.7,12
Profits,2,4
Achats,8,9
```

Hélas, cette technique ne marche pas avec un Excel français, qui utilise des points-virgules au lieu des virgules : la raison est que la virgule est utilisée comme séparateur décimal en français. Si vous n'avez pas un Excel US, je vous conseille donc trois méthodes pour faire un fichier CSV :

a) Tapez le texte du fichier dans GUSEK : séparez les champs successifs par des virgules, sans espaces (comme ci-dessus). Enregistrez le fichier obtenu avec le suffixe ".csv".

b) Tapez vos tableaux dans une feuille Excel ou calculez-les avec VBA. Enregistrez le fichier Excel obtenu au format CSV, puis ouvrez-le avec GUSEK. Si vous avez des nombres à décimales comme "3,75", remplacez d'abord toutes les virgules par des points avec la commande "Search/Replace". Ensuite, remplacez tous les points-virgules par des virgules et enfin enregistrez le fichier.

c) Si vous calculez des tableaux avec un programme C, vous pouvez générer un fichier CSV dans votre programme. Voici un exemple de code pour deux tableaux de 10 éléments : un tableau d'entiers T1 et un tableau de réels T2. Ici on n'utilise pas ici l'indice 0 des tableaux. Les deux tableaux sont initialisés n'importe comment, juste pour avoir des contenus. On veut écrire l'indice dans une première colonne, puis la valeur de T1, et enfin la valeur de T2 avec deux décimales :

```
int i;
int T1[6];
float T2[6];
FILE* F;
for (i = 1; i <= 10; i++){
   T1[i] = 10*i;
   T2[i] = sin(i);
};
F = fopen ("mon-fichier.csv","w");
for (i = 1; i <= 10; i++) fprintf (F,"%d,%d,%f.2\n",i,T1[i],T2[i]);
fclose (F);
```

8.2 Instruction "table"

Considérons deux tableaux qui donnent le nombre de jours et la température moyenne pour les mois de janvier à mars (indices 1 à 3). On peut les initialiser en parallèle dans la section "data" :

```
param jours {i in 1..3};
param temp  {i in 1..3};
data;
param : jours, temp :=
1, 31,  0
2, 28,  5
3, 31, 12;
```

Les virgules sont facultatives. Pour initialiser avec une table, tapez avec GUSEK le fichier suivant puis enregistrez-le sous le nom "table-mois.csv" :

```
mois,jours,temp
1,31,0
2,28,5
3,31,12
```

Il y a quatre différences avec la section "data" : les virgules sont obligatoires, les espaces sont interdits, les colonnes de la table doivent tous avoir un nom (comme en Access) et il n'y a plus à la fin le point-virgule qui devait terminer l'instruction "param". Ici, on a choisi comme noms de colonnes les noms des tableaux dans le modèle car GUSEK va faire la correspondance automatiquement. La colonne "mois" sert seulement à nommer la colonne des indices : elle ne correspond à rien dans le modèle. La table est lue comme suit :

```
param jours {i in 1..3};
param temp  {i in 1..3};
table ma_table IN "CSV" "table-mois.csv" : [mois], jours, temp;
display jours, temp;
```

"ma_table" sert à nommer la table dans le modèle mais le nom n'a pas d'utilisation dans la version actuelle de GMPL. "IN" signifie "input" : la table va être lue dans un fichier. On peut aussi écrire une table dans un fichier avec "OUT" (output), voir le manuel GMPL. "CSV" signifie que la table est au format CSV (le manuel décrit d'autres formats). La commande "table" donne ensuite le nom du fichier puis la liste des noms de colonnes tels qu'ils sont donnés dans le fichier.

Il doit toujours y avoir un premier nom entre crochets, appelé "colonne-clé" : c'est le nom de la colonne des indices dans le fichier, son contenu va être vérifié par GUSEK. Si les noms de tableaux sont différents dans le modèle et dans le fichier, on définit la correspondance avec un tilde "~". Par exemple, si dans le fichier les noms de colonnes sont "MOIS", "JOURS", "TEMP", on peut écrire :

```
table ma_table IN "CSV" "table-mois.csv" : [MOIS], jours ~ JOURS, temp ~ TEMP;
```

Ce qui indique à GUSEK que les contenus des colonnes "JOURS" et "TEMP" doivent être chargés dans les tableaux "jours" et "temp", respectivement. Les colonnes peuvent être dans n'importe quel ordre dans le fichier. Il peut même y avoir d'autres colonnes : elles seront ignorées à la lecture. La technique est identique pour des indices symboliques. Reprenons l'exemple de la page 11 :

```
set mois;
param ventes {i in mois};
param profit {i in mois};
…
data;
set mois := Jan, Fev, Mar;
param : ventes, profit :=
Jan 33 5
Fev 49 7
Mar 66 6;
```

Pour initialiser "ventes" et "profit" avec une table, taper avec GUSEK un fichier "test.csv" sans espaces, contenant :

```
Mois,Ventes,Profit
Jan,33,5
Fev,49,7
Mar,66,6
```

On peut ensuite le lire avec :

```
table autre_table IN "CSV" "test.csv" : [Mois], ventes ~ Ventes, profit ~ Profit;
```

On a vu dans la section 5 qu'on pouvait définir l'ensemble "mois" en même temps :

```
data;
param : mois : ventes, profit :=
Jan 33 5
Fev 49 7
Mar 66 6;
```

Cette technique est possible avec une table, sans changer le fichier CSV. La différence est la flèche, qui indique que les indices trouvés dans le fichier doivent être rangés dans l'ensemble "mois".

```
table tab IN "CSV" "test.csv" : mois <- [Mois], ventes ~ Ventes, profit ~ Profit;
```

Enfin, on peut avoir des fichiers avec plusieurs indices : les noms des colonnes correspondantes doivent être listés entre crochets. Page 12, on a vu comment initialiser en parallèle un ensemble d'arcs *A* puis deux tableaux *Capa* et *Cost* indicés par des arcs :

```
set A := (1,3) (4,2) (7,1);
param : Capa Cost :=
1 3 19 24
4 2 41 35
7 1 22 57;
```

Pour les initialiser à partir d'une table, créer un fichier "graphe.csv" sans espaces :

```
i,j,Capa,Cost
1,3,19,24
4,2,41,35
7,1,22,57
```

Qui peut être lu avec :

```
table tab IN "CSV" "graphe.csv" : [i,j], Capa, Cost;
```

Les colonnes "i" et "j" seront utilisées pour récupérer les indices et ranger les valeurs associées dans les tableaux *Capa* et *Cost*. On peut utiliser le même procédé pour lire une ou plusieurs matrices en parallèle : il faut deux colonnes pour les indices et une colonne pour les valeurs de chaque matrice.

Pour initialiser en même temps *A* (le "set A :=" devenant inutile), il suffit d'écrire :

```
table tab IN "CSV" "graphe.csv" : A <- [i,j], Capa, Cost;
```

Pour terminer cette section, on peut combiner des tables et une section "data". Dans ce cas, il faut savoir que les tables sont chargées par GUSEK avant d'examiner la partie "data". On peut donc utiliser des paramètres provenant des tables pour initialiser des paramètres dans la section "data".

Certains logiciels de modélisation très chers comme OPL Studio (basé sur le solveur Cplex) peuvent lire de vrais fichiers Excel (.xls). Ce n'est pas encore le cas de GUSEK.

Dans les logiciels commerciaux contenant des programmes linéaires (comme certains modules de SAP), les données sont toujours sous forme de fichiers-tables. Ces fichiers sont extraits automatiquement de bases de données pour éviter des erreurs de saisie. L'extraction peut se faire par des commandes SQL, des programmes en VBA etc.

9. QUELQUES SUBTILITES DE GMPL POUR FINIR

Quelques trucs pas évidents qui me sont arrivés…

9.1 Ignorer des valeurs dans un "printf"

Un cas fréquent consiste à écrire des valeurs selon une condition, par exemple les éléments supérieurs à 5 d'un vecteur T. On peut utiliser le "if" de GMPL, à ne pas confondre avec le "if" de C ou VBA : le "if" de GUSEK est semblable à la formule "if" d'Excel et renvoie une valeur selon une condition logique. L'instruction suivante affiche hélas des zéros pour les autres éléments :

```
printf {i in 1..n} "%d ", if T[i] > 5 then i;
```

L'astuce est de sélectionner avec le "if" la chaîne de format au lieu de la valeur à afficher :

```
printf {i in 1..n} if T[i] > 5 then "%d " else "", j;
```

Mais une méthode plus générale est d'utiliser la possibilité de préciser une condition sur les indices. C'est possible dans les équations du PL (sommes par exemple) mais GUSEK accepte aussi des conditions dans les "printf" :

```
printf {i in 1..n :  T[i] > 5} "%d ", j;
```

9.2 Tableau dont les éléments font partie d'un ensemble

On peut définir un tableau dont les éléments appartiennent à un ensemble. Soit un ensemble de villes, un ensemble de quartiers, et un tableau "ville" qui indique la ville pour chaque quartier :

```
set villes;
set quartiers;
param ville {i in quartiers} in villes;
```

Si on compile sans "data", GUSEK ne signale aucune erreur de syntaxe avec ces instructions, il va juste dire qu'il manque les données. Mais si on ajoute la partie "data" suivante, GUSEK dit qu'il veut des valeurs numériques entières pour les éléments du tableau "ville"!

```
data;
set villes    := "Paris", "Lyon", "Lille";
set quartiers := "Cité", "Concorde", "Part-Dieu", "Opéra";
param ville :=
"Cité"       "Paris"
"Concorde"   "Paris"
"Part-Dieu"  "Lyon"
"Opéra"      "Lille";
```

Le problème vient du fait que pour GUSEK les "param" sans préciser le type de données sont des nombres réels par défaut. Les éléments de "villes" sont bien des noms, mais GUSEK est incapable d'en déduire que c'est pareil pour les éléments du tableau "ville" : le "in villes" ne suffit pas, il faut préciser que les éléments de "ville" sont de type chaîne (mot-clé "symbolic") :

```
param ville {i in quartiers} symbolic in villes;
```

9.3 Bugs

GUSEK contient très peu de bugs. J'en ai détecté quand même un : on obtient parfois une valeur incorrecte si on fait un "display" ou un "printf" faisant référence au nom de la fonction-objectif ou à un nom de contrainte. Par exemple, le display suivant donne parfois un résultat faux :

```
minimize z : sum {i in 1..n} c[i]*x[i];
…
solve;
display "Objectif = ",z;
```

On peut s'en apercevoir car GUSEK indique toujours la valeur correcte à la fin de son compte-rendu d'exécution, dans la fenêtre de droite. Si cela vous arrive, il suffit de répéter dans le display la formule de l'objectif :

```
display "Objectif = ", sum {i in 1..n} c[i]*x[i];
```

9.4 Erreurs à l'exécution

"LP has unbounded solution" ou "Dual is infeasible". Cette erreur signale l'absence d'optimum fini car le polyèdre n'est pas borné. Il manque en général une contrainte. 9 fois sur 10, l'erreur vient du fait qu'on a oublié de dire que les variables doivent être positives ou nulles, dans un PL en minimisation.

"Cannot find a feasible solution". GUSEK ne trouve pas de solution réalisable. La cause peut être très difficile à trouver. Dans les problèmes de transport, on a par exemple des demandes supérieures aux disponibilités ou le réseau est déconnecté à cause d'erreurs de saisie (il manque des arcs). Si après avoir vérifié les données l'erreur persiste, il est probable que le problème soit trop contraint. Par exemple, vous avez tapé >= au lieu de <=. Si l'erreur persiste toujours, essayez de détecter la contrainte fautive en mettant un par un chaque groupe de contraintes en commentaires /* … */ et en re-testant : si le PL devient faisable, les contraintes en commentaires sont responsables.

"Out of memory". Actuellement, GUSEK réserve seulement une zone de mémoire de 2 GB pour ses calculs, même si vous avez plus de mémoire. Un PL trop gros peut remplir la mémoire. En général, le PL tient en mémoire, mais c'est l'arborescence des calculs (pour un PLNE) qui remplit lentement la mémoire. Le phénomène peut apparaître parfois après des minutes, voire des heures de calcul. GUSEK étant un logiciel public, les sources sont disponibles et il doit être possible de modifier le code pour agrandir la taille de la zone de calcul. Mais ça me parait très sportif car il faut recompiler le logiciel complet!