Tutoriel de MODELIB

Quentin Lequy et Romain Gaucher

21 mai 2005

Première partie

Les Classes utiles de *Modelib*

Merci d'avoir choisi MODELIB comme Modeleur de Programmes Linéaires. Il faut tout de suite préciser que la syntaxe de ce modeleur a été calquée sur celle de OPL de l'entreprise ILOG.

Dans un premier temps, nous allons détailler les différentes classes que contient MODELIB, puis nous allons voir des exemples de programmes linéaires codés en *C++*, et finalement nous listerons les prototypes des fonctions décrites dans ce tutoriel.

Dans tout ce document, les classes seront notées `Classes` et les fonctions `Fonctions`. Les arguments par défaut des constructeurs seront entre crochet et les fonctions n'auront ni argument, ni type de retour, mis à part dans la troisième partie.

On fera aussi régulièrement référence à la classe `Model` qui est le coeur du modeleur mais qui est définie seulement en dernier lieu dans la première partie. Ceci vient du fait qu'il n'est pas nécessaire de connaître la classe `Model` pour utiliser les outils de MODELIB mais qu'il est préférable de connaître les outils de MODELIB pour comprendre comment fonctionne la classe `Model`.

Table des matières

1 Les classes *Float*, *Int*, *Bool*

1.1 Description

Ce sont les classes représentant des scalaires. Elles n'ont pas d'autre utilité que de faciliter la construction des expressions grâce à la surcharge de leurs constructeurs. Les scalaires servent à coefficienter les variables dans les expressions. Ils dérivent de la classe `Num`

1.2 Utilisation

1.2.1 Construction

Il est possible d'utiliser le constructeur de type Num([scalaire]) ou scalaire peut être de type du C++ :

- `float`
- `double`
- `unsigned`
- `int`

Suivant le type de la classe `Num`, la convertion est faite automatiquement. Le type par défaut d'une classe `Num` est réel.

1.2.2 Méthodes

Les méthodes sont les accesseurs de type, `GetType` et `SetType`, et les accesseurs de valeurs `GetValue` et `SetValue`

1.2.3 Astuces techniques et exemples d'utilisation

Il est possible de reattribuer une valeur à la variable grâce à l'`operator=` ou en utilisant le constructeur de copie.

1.3 Ce qu'il faut savoir d'autre

Il est possible de s'en passer des classes Num au moins apparemment, mais en réalité tous les scalaires sont convertis en `Num` avant d'être intégrer à des expressions affines.

2 Les classes *FloatArray*, *IntArray*, *BoolArray*

2.1 Description

Ce sont les classes représentant des vecteurs de scalaires. Elles n'ont pas d'autre utilité que de faciliter la construction des expressions. Les vecteurs de scalaires servent à coefficienter les vecteurs de variables dans les expressions grâce au produit salaire. Ils dérivent de la classe `NumArray`

2.2 Utilisation

2.2.1 Construction

Il est possible d'utiliser le constructeur de type NumArray([std : :vector< scalaire >]) ou scalaire peut être de type du C++ :

- `float`
- `double`
- `unsigned`
- `int`

Suivant le type de la classe `NumArray`, la conversion est faite automatiquement. Le type par défaut d'une classe `NumArray` est réel.

2.2.2 Méthodes

Les méthodes sont les accesseurs de type, `GetType` et `SetType`, et les accesseurs aux éléments tel que l'`operator[]` ou la fonction `Get`

2.2.3 Astuces techniques et exemples d'utilisation

Il est possible d'additionner, soustraire, ou multiplier par un scalaire des NumArray.

```
Model modele;

std:vector<float> temp;
temp.push_back(1.5);
temp.push_back(0.75);
temp.push_back(2.2);

FloatArray v1(temp);      // v1 = [1.5, 0.75, 2.2]
FloatArray v2(3,0.25);    // v2 = [0.25, 0.25, 0.25]

FloatArray v3 = v1-4*v2; // v3 = [0.5, -0.25, 1.5]
```

2.3 Ce qu'il faut savoir d'autre

Il faut faire attention aux types que l'on utilise pour bien avoir les données que l'on veut. Le transtypage est certes automatisé mais peut conduire à des incohérences si les opérations sur les `NumArray` sont faites sans précaution, surtout concernant les `BoolVar`. Les `NumArray` ne sont qu'une aide, il bien sûr possible de s'en passer.

3 Les classes *FloatVar*, *IntVar* et *BoolVar*

3.1 Description

Ces trois classes dérivent de la classe `NumVar`. Cette classe représente les variables. Les variables peuvent être de trois types : continues, entières ou booléennes. Pour rajouter une variable au programme linéaire, il suffit d'instancier une des NumVars. Attention, le constructeur des `NumVar` requiert absolument une référence sur une classe `Model`. Elles peuvent être ajoutées dans une expression linéaires `Expr` ou bornées par le biais d'une contrainte de borne.

3.2 Utilisation

3.2.1 Construction

Les constructeurs des NumVars sont les suivants :
- `BoolVar( Model [, nom] )`
- `IntVar(Model[, borne inférieure][, borne supérieure][, nom])`
- `IntVar(Model[, nom])`
- `FloatVar(Model[, borne inférieure][, borne supérieure][, nom])`
- `FloatVar(Model[, nom])`

Pour spécifier au constructeur le domaine de définition, la constante `Infinity` sert d'infini. Par défaut le domaine de définition d'une variable est $[0.0, +Infinity]$. Le nom par défaut est $unknow\{id\}$ où id est le numéro de la variable

Une fois instanciée, la variable ne peut être détruite qu'à partir du Model

Pour instancier une classe sans ajouter de variable, il faut utiliser le constructeur :
`NumVar(Model *, VarId)`

Enfin, l'appel du constructeur par défaut des NumVar créer une variable neutre

3.2.2 Méthodes

Beaucoup de caractéristiques d'une variable peuvent être extraites grâce aux NumVars
- Son nom par la méthode `GetName`
- Sa borne inférieure par la méthode `GetLowerBound`
- Sa borne inférieure par la méthode `SetLowerBound`
- Sa valeur après résolution par la méthode `GetValue`
- Son type par la méthode `GetType`
- Un pointeur sur son Model par la méthode `GetModel`
- Son identifiant dans le Model par la méthode `GetVarId`

Beaucoup de caractéristiques des variables peuvent être redéfinies grâce aux NumVars :
- Son nom par la méthode `SetName`
- Sa borne inférieure par la méthode `SetLowerBound`
- Sa borne inférieure par la méthode SetLowerBound
- Son type par la méthode `SetType`

3.2.3 Astuces techniques et exemples d'utilisation

Pour dupliquer une variable sans en recréer une, il suffit de faire :

```
Model modele;                          //déclarations du modèle

FloatVar x(modele,0,Infinity,"x")      //création de la variable
NumVar autre_x(&modele,x.GetVarId())   //duplication de la variable

cout << "Nom = " << autre_x.GetName() << endl;  // affiche "Nom = x"
```

Pour redéfinir les bornes d'une variable,il suffit de taper :

```
float lb = -2.0;
float ub = 10;
modele.Add( lb <= x < ub ); // ou modele.Add( ub > x >= lb );
```

La classe Nom permet créer des noms de variable plus facilement.

3.3 Ce qu'il faut savoir d'autre

Une variable est définie dans le modèle par un identifiant. Il est préférable de lui donner un nom même si ce n'est pas nécessaire. Pour supprimer une variable, il faut récupérer cet identifiant. Détruire une variable est une action coûteuse et peut être souvent évitée. La valeur d'une variable ne change qu'après résolution du programme linéaire. Il est possible d'avoir accès de changer le caractère strict des bornes d'une variable par les méthodes `IsLowerBoundStrict`, `SetLowerBoundStrict`, `IsUpperBoundStrict` et `SetUpperBoundStrict`.

4 Les classes *FloatVarArray*, *IntVarArray* et *BoolVarArray*

4.1 Description

Ces classes dérivent de la classe `NumVarArray` et sont simplement des vecteurs de `NumVar`. Le but des `NumVarArray` est double : générer automatiquement des variables semblables et faciliter l'utilisation des `NumVar`. En effet dans les programmes linéaires, les variables sont souvent indexées sur des sous-ensembles de N

4.2 Utilisation

4.2.1 Construction

Il est possible de construire des `NumVar` de deux façon : classiquement à partir des constructeurs automatisés :

```
BoolVarArray(Model[, taille ][, nom][, fin de nom])
IntVarArray(Model [, borne inf][, borne sup][, nom][, fin de nom])
FloatVarArray(Model[, taille][, borne inf][, borne sup][, nom][, fin de nom])
```

ou à partir d'un vecteur de `NumVar` déjà existant par le constructeur :
`NumVarArray( const std::vector<NumVar> \& );`

Dans le premier cas, l'indice de la variable dans le tableau est rajouté entre le nom et la fin du nom, ceux-ci étant bien sûr optionnels. Pour spécifier au constructeur le domaine de définition qui vaudra pour toutes les variables, la constante `Infinity` sert d'infini. Par défaut le domaine de définition des variables est $[0, +Infinity]$. Il est conseillé de donner un nom au vecteur même si ce cas est géré.

4.2.2 Méthodes

Les méthodes des `NumVarArray` sont principalement des accesseurs. En effet il est possible :
- de connaître sa taille par la méthode `Size`
- d'accéder à un de ses éléments par la méthode `Get`
- d'accéder en lecture à un de ses éléments par l'`operator[]`
- de connaître le type par la méthode `GetType`

4.2.3 Astuces techniques et exemples d'utilisation

Il est possible de créer des expressions `Expr` grâce à la fonction `Sum` qui renvoie la somme des variables du vecteur passé en paramètre. Il est aussi possible de faire des produits scalaires avec des vecteurs de scalaires, ce qui retourne aussi une expression.

```
Model modele;

std:vector<float> temp;
temp.push_back(1.5);
temp.push_back(0.75);
temp.push_back(2.2);

FloatArray v1(temp);              //v1 = temp (conversion en FloatArray)
FloatArray v2(3,0.25);            //v2 = [0.25, 0.25, 0.25]
FloatVarArray x(modele,3,0,Infinity,"x");

Expr e = ((v1-4*v2)*x);
cout << Sum(x) + e << endl;

//Sortie:
// 1.5x0 + 0.75x1 + 2.2x2
```

4.3 Ce qu'il faut savoir d'autre

Il n'est pas possible de rajouter des variables dans le vecteur une fois qu'il a été créé. Le but des `NumVar` est de faciliter la création d'expressions affines `Expr`. Il n'est pas possible de créer des expressions directement comme pour les variables. Il faut toujours soit passer par la fonction `Expr` soit par le produit scalaire.

5 La classe *FloatVarMatrix*, *IntVarMatrix* et *BoolVarMatrix*

5.1 Description

Les NumVarMatrix sont les extensions à deux dimensions des NumVarArray à deux dimensions. Leur principale utilité est la récupération des lignes et des colonnes sous la forme de NumVarArray.

5.2 Utilisation

5.2.1 Construction

Il y deux façon de construire les NumVarMatrix. La traditionnelle est le constructeur :

```
FloatVarMatrix(  Model
                 [, nombre de lignes,]
                 [, nombre de colonnes]
                 [, borne inférieure]
                 [, borne supérieure]
                 [, chaine de début ]
                 [, chaine de milieu ]
                 [, chaine de fin ] )
```

La deuxième à partir d'un vecteur de pointeurs sur NumVarArray représentant les lignes de la matrice. `NumVarMatrix( const std::vector<NumVarArray *> &)` Les arguments des constructeurs sont similaires à ceux des `NumVarArray` mis à part la longueur et la largeur du tableau. Le premier argument est bien sûr obligatoire, les autres sont facultatifs mais il est conseillé de les utiliser, tout du moins jusqu'au nom.

5.2.2 Méthodes

Les méthodes utiles de `NumVarMatrix` sont :
- `GetNbRows` qui permet de récupérer le nombre de lignes de la matrice.
- `GetNbCols` qui permet de récupérer le nombre de colonnes de la matrice.
- `GetRow` qui permet de récupérer une ligne de la matrice sous forme d'un `NumVarArray`.
- `GetCol` qui permet de récupérer une colonne de la matrice sous forme d'un `NumVarArray`.
- l'`operator()` qui permet de récupérer un élément dans la matrice en fonction de ses coordonnées.
- `Get` qui permet de récupérer un pointeur sur un un élément dans la matrice.
- `GetType` qui permet de récupérer le type de variable de la matrice.

5.2.3 Astuces techniques et exemples d'utilisation

Il est plus facile et rapide de récupérer une ligne qu'une colonne car la matrice est stockée sous forme d'un vecteur de lignes.

```
Model modele; //on déclare le Model

FloatVarMatrix A(modele,2,3,0,Infinity,"M","_"); // on crée une matrice de 2 →
    ↪ lignes et 3 colonnes

for( unsigned i = 0; i < A.GetNbRows(); ++i) //Pour chaque ligne
  cout << Sum(A.GetRow(i)) << endl;          //On affiche la somme (voir →
     ↪ NumVarArray)

// Sortie :
// M0_0+M0_1+M0_2
// M1_0+M1_1+M1_2
```

5.3 Ce qu'il faut savoir d'autre

Pour générer un tableau avec plus de deux dimensions, il existe une classe appelée *NumVarHyperMatrix* basée sur des templates récursifs et dont une utilisation peu prudente peut conduire à des erreurs de segmentation difficilement détectable. Bien que la classe existe, il est déconseillé de l'utiliser sans une certaine connaissance de sa structure. C'est pour cette raison que cette classe n'apparaît pas dans le tutoriel.

6 Les classes templates *Ensemble*, *Partie* et *Propriete*

6.1 Description

La classe `Ensemble` est un conteneur représentant les ensembles mathématiques. La classe `Partie` représente une partie d'un ensemble et ne peut exister sans être rattachée à un ensemble. Les classes `Ensemble` et `Partie` permettent de réaliser sur un ensemble de données toutes les opérations ensemblistes de base. Néanmoins la classe `Ensemble` ne représente que des ensembles non ordonnés tels les arcs d'un graphe. Construire une classe `Partie` se fait généralement suivant une certaine propriété qui différentie les éléments d'un ensemble : c'est le rôle du type de classe `Propriete`.

6.2 Utilisation

6.2.1 Construction

Que ce soit les classes `Ensemble`, `Partie` ou encore `Propriete`, leur construction nécessite de connaitre et de savoir utiliser les notions de généricité en *C++*. Il y a deux façon de construire une classe `Ensemble` :

- à partir d'une liste d'éléments : si T est le type des élements,
  il suffit d'appeler `Ensemble( std::list<T> base)` pour créer un `Ensemble<T>`
- en ajoutant les éléments un à un dans un ensemble initialement vide grâce à la fonction `Add`

Cependant cette dernière méthode doit être utilisée avec prudence : les parties créées avant l'appel de la fonction `Add` ne sont pas réactualisées. De plus, il n'est pas possible de supprimer un élément d'un `Ensemble` lorsque celui-ci est construit, il est censé être immuable : toutes les opérations ensemblistes se feront sur les classes `Partie`. Il a trois façon de construire une classe `Partie` :

- à partir d'un `Ensemble` en prenant tout ses éléments grâce à la fonction `all` de `Ensemble`
- à partir d'un `Ensemble` en prenant les éléments répondant à une `Propriete` à la fonction `partie` de `Ensemble`
- en utilisant le constructeur de `Partie` qui prend comme paramètre un `Ensemble`

Pour construire une partie à partir d'un ensemble, il faut donc utiliser une `Propriete`. Cette classe étant abstraite, il faut donc coder une classe héritant de `Propriete` surchargeant l'`operator()` virtuel.

6.2.2 Méthodes

Les méthodes de la classe `Ensemble` sont :

- `partie` qui permet donc de créer une partie à partir d'une `Propriete`,
- `all` qui permet de créer une partie contenant tous les éléments de l'ensemble
- `Get` qui permet de récupérer un élément de l'ensemble dans l'ordre de leur création,
- `Size` qui permet de récupérer la taille de l'`Ensemble`
- `Add` qui permet d'ajouter un élément à l'`Ensemble`.

Les méthodes de la classe `Partie` sont :

- `_union` qui permet de faire une union avec une autre Partie du même Ensemble,
- `_inter` qui permet de réaliser une intersection avec une autre Partie,
- `_compl` qui permet d'extraire le complémentaire d'une autre Partie dans cette Partie,
- `GetIds` qui permet de récupérer la liste des identifiants des éléments de la Partie,
- `Get` qui permet de récupérer une liste des éléments de la Partie.
- `Size` qui permet de retourner la taille de la Partie

Les méthodes externes aux classes Partie et Ensemble sont les surcharges d'opérateurs suivantes

- `&` qui permet de faire une intersection entre deux parties du même Ensemble,
- `|` qui permet de faire une union entre deux parties du même Ensemble,
- `/` qui permet d'extraire le complémentaire d'une Partie dans une autre Partie,
- `/` qui permet d'extraire le complémentaire d'une Partie dans un Ensemble,
- `<<` qui permet d'afficher un Ensemble ou une Partie

6.2.3 Astuces techniques et exemples d'utilisation

Pour construire un ensemble à partir d'un fichier où les différents éléments sont mis à la suite, il est possible de faire appelle à l'`operator>>` qui récupérera un SEUL élément dans le fichier si l'`operator>>` est surchargé pour le type de l'élément de l'`Ensemble`. Il faudra donc faire une boucle sur le nombre d'éléments dans le fichier saisit par exemple au début du fichier.

```
Ensemble<unsigned> E;          //on declare un ensemble d'entiers

//on y ajoute les entiers de 0 à 9 qui sont bien tous différents
for(unsigned i = 0; i < 10; ++i)
   E.Add(i);

struct EstPair : public Propriete<unsigned> //On crée la propriete de parite
{
 //Rmq: le const & permet de respecter le prototype de l'operateur virtuel ()
   bool operator() ( const unsigned  &n) const
   {
       return (n%2==0);
   }
};

//On déclare une partie qui est celle des nombres pairs
Partie<unsigned> A = E.partie( EstPair() );
//Rmq : EstPair est vu comme un Foncteur

cout << E << endl
     << A << endl
     << E/A << endl;

// Sortie :
// {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
// {0, 2, 4, 6, 8}
// {1, 3, 5, 7, 9}
```

6.3 Ce qu'il faut savoir d'autre

Important : Ni la fonction Add, ni le constructeur à partir de liste n'assure l'unicité des éléments de l'`Ensemble`. C'est à l'utilisateur de considérer que les éléments de l'`Ensemble` sont uniques. En cas de duplication, les éléments identiques seront considérés comme différents.

Une Partie a besoin d'un ensemble pour exister. Il n'est pas possible de faire des unions, des intersections d'Ensemble. En effet, il faut voir l'ensemble comme une base de travail pour manipuler des parties que l'on pourra considérer comme des ensembles au sens mathématique. Comme elle a été pensée pour les graphes, la gestion des Ensembles et des Parties n'est pas plus poussée, il n'est pas possible de faire des produits cartésiens ou de créer des ensembles non discrets par exemple.

7 Les classes *FloatVarFamily*, *IntVarFamily* et *BoolVarFamily*

7.1 Description

Les classes NumVarFamily sont en fait des familles de variables indexées sur un Ensemble de Nuplet, c'est-à-dire que chaque variable est associée à un n-uplet de l'ensemble. La classe Nuplet représente tout simplement un couple, un triplet ou plus généralement un n-uplet d'un certain type d'élément, des entiers par exemple. Mathématiquement on peut considérer :

Soit E un ensemble d'éléments.
Pour $n > 0$ on définit un ensemble $I \subset E^n$.
$\forall y = (x_0, ..., x_n) \in I$, on considére la variable $v_y = v_{x_0,...,x_n}$.
La classe NumVarFamily va représenter la famille $F = \{v_y, y \in I\}$

7.2 Utilisation

7.2.1 Construction

La construction d'une NumVarFamily$< N, T >$, où $N \in \mathbb{N}$ et T est un type, passe uniquement par l'appel des constructeurs :

```
BoolVarFamily(Model[, Ensemble ][, format du nom])
IntVarFamily(Model[, borne inf][, borne sup][, Ensemble ][, format du nom])
FloatVarArray(Model[, borne inf][, borne sup][, Ensemble ][, format du nom])
```

La création des noms des variables fonctionne différemment des autres conteneurs de variables. En effet, comme les indices des variables peuvent être de n'importe quel type, le nom des variables ne peut pas être défini comme pour les variables des classes `NumVarMatrix` par exemple. Il va être créé suivant un format ressemblant à celui du fameux printf utilisée en C mais en légèrement différent.

Il sera sous la forme d'une chaîne de caractères comprenant une répitition du caractère spécial "%" qui devra apparaître pour chaque x_i, $0 \leq i \leq n$. Pour une famille indicées un `Ensemble` de couples, on devra avoir une chaîne du type "nom_de_la_variable_%_intermédiaire_%_terminaison".

7.2.2 Méthodes

Les trois méthodes intéressantes de la classe `NumVarFamily` sont :
- `GetFamilyPart` qui permet, à partir d'une `Partie` de l'`Ensemble` des indices ou d'une `Propriete` sur cet `Ensemble`, de récupérer un `NumVarArray` qui contient les variables correspondant aux indices contenus dans la Partie.
- `Get` qui permet d'obtenir une variable, son argument étant un entier correspondant à l'ordre de création de la variable lors de création de la famille.
- `Size` qui permet de récupérer la taille de la famille.

7.2.3 Astuces techniques et exemples d'utilisation

La méthode `GetFamilyPart` est utile pour faire des sommes sur des parties de l'`Ensemble` d'indices. En effet, puisqu'elle retourne une classe `NumVarArray`, il est ensuite possible des opérations sur le retour comme la somme ou le produit scalaire. Il faut néanmoins faire attention aux dimensions. Il faudra quand même créer une `Propriete`, comme la `Propriete Omega` définie ultérieurement.

La méthode `Get` peut être aussi très utile pour la manipulation des classes `NumVarFamily`, il suffit juste être conscient de l'ordre dans lequel ont été instanciées les variables qui est le même que l'ordre dans lequel a été instancié l'`Ensemble` des indices. Or l'ordre d'instanciation d'un élément dans un `Ensemble` est soit son ordre dans la liste de construction, soit son ordre dans le fichier de données duquel il provient, soit l'ordre dans lequel il a été passé à la méthodes `Add`.

Pour l'utilisation d'un Ensemble d'indices, il faut faire attention à l'écriture des templates lors de la déclaration. En effet il est conseillé de mettre un espace avant et après les classes contenues pour ne pas avoir d'ambiguïté avec l'`operator>>`.

```
    // Couple est une redéfinition de Nuplet<2,unsigned>
    std::list< Couple > listTmp;
    Couple n;
    //on crée une liste de Couples
    for(unsigned i =0; i < 10; ++i)
    {
      n[0] = i%5;
      n[1] = i%3;
      listTmp.push_back(n);
    }

    //On crée un ensemble à partir de cette liste
    Ensemble< Couple > E(listTmp);

    struct OneIsNull : public Propriete<Couple>
    {
      bool operator() ( const Couple  &nCouple) const
      {
        return (nCouple[0] == 0 || nCouple[1] == 0);
      }
    };

    Model modele;

    FloatVarFamily<2,unsigned> F(modele,-Infinity,2.0f,E,"f%_%");

    NumVarArray x = F.GetFamilyPart( OneIsNull() );

    cout << Sum(x) << endl;

//Sortie :
// f0_0+f3_0+f0_2+f1_0+f4_0
```

7.3 Ce qu'il faut savoir d'autre

L'ensemble des indices est un attribut public de NumVarFamily. Il est donc possible en théorie de le modifier à tout instant. Il faut cependant savoir qu'une fois instanciée, la classe NumVarFamily est immuable au niveau de ses variables, à savoir qu'aucun ajout n'est possible.

Il est préférable de surcharger l'opérateur `<<` pour le type servant à l'indiçage. En effet, celui-ci est nécessaire dans la création du nom des variables. Attention cependant à ce que les noms générés des variables des soient corrects, c'est-à-dire qu'ils respectent la norme pour les noms en *C++*.

Il faut finalement faire bien attention aux divers types utilisés car `NumVarFamily`, et les classes qui lui sont rattachées, sont des classes génériques. Il pourra donc se produire un nombre d'erreurs conséquent pour une simple substitution d'un **int** en **float**.

8 Les classes *Expr* et *ConstraintBuilder*

8.1 Description

La classe `Expr` représente les expressions affines, c'est-à-dire les combinaisons linéaires de variables représentée par NumVar. On a donc :
Si $\{v_i, i \in I\}$ un ensemble de variables et $\{a_i \in \mathbb{R}, i \in I\}$,
$Expr \Leftrightarrow \sum_{i \in I} a_i v_i + cste$

La classe `ConstraintBuilder` représenter les inégalités et égalités linéaires. Elle est basée sur plusieurs opéarteurs : $=, \leq, \geq, <, >$. C'est un enchaînement d'`Expr` et d'opérateurs qui définissent en fait les contraintes linéaires à ajouter au `Model`.

Le fonctionnement de ces deux classes repose sur la surcharge des opérateurs en *C++*. La création et la manipulation va en fait être un jeu d'écriture au sein même du code, de façon assez intutive.

8.2 Utilisation

8.2.1 Construction

Il y a deux façons de construire une `Expr` ou un `ConstraintBuilder`. La première est explicite : on va appeler explicitement un constructeur Expr et on va rajouter à l'Expr les termes de la somme un à un. La deuxième est implicite : on va écrire dans le code la combinaison comme si on l'écrivait sur du papier. Ces deux façons peuvent bien sûr être mélangées suivant la situation. Les constructeurs explicites d'Expr vont permettre de construire en fait le premier terme de la somme. Ils sont `Expr()`, `Expr( NumVar )`, `Expr( float )`, `Expr( double )`, `Expr( int )`, `Expr( unsigned )`, `Expr( Num )`, `Expr( Num, NumVar )`.

La façon implicite de déclaration va utiliser les opérateurs $*$ entre un scalaire et une variable, $+$ entre deux `Expr`, $-$ entre deux `Expr`, $*$ entre un scalaire et une `Expr`, et finalement $*$ entre un vecteur de variable et un vecteur de scalaire.

Pour coupler les deux méthodes de construction, on peut utiliser les opérateurs +=, -= ou *= pour manipuler les expressions, ceux-ci ayant le même rôle que +, - et *, sauf que le terme de gauche de l'opérateur est l'expression courante.

Pour construire un `ConstraintBuilder` il suffit d'utiliser soit le constructeur `ConstraintBuilder( Expr, operateur, Expr)`, soit plus simplement utiliser les operateurs *C++* suivant entre deux `Expr` : ==, < = , >=, <, >.

8.2.2 Méthodes

Les méthodes propres à `Expr` et à `ConstraintBuilder`. Les méthodes de `Expr` sont :
- `Size` qui permet de connaître le nombre de terme de la somme,
- `IsNum` qui permet de savoir si l'expression ne se réduit qu'à un scalaire,
- `IsVar` qui permet de savoir si l'expression ne se réduit qu'à une variable,
- `GetConstant` qui permet de récupérer la partie constante de l'expression,
- `Evalue` qui permet de récupérer la valeur de l'expression,
- `ASS` qui permet d'ajouter un terme à l'expression de façon directe.

La seule méthode de `ConstraintBuilder` est `IsBound` qui permet de savoir si la contrainte construite est une borne.

Il existe deux autres fonctions importantes : la fonction `Sum`, qui renvoie sous forme d'`Expr` la somme des éléments d'un `NumVarArray`, et l'`operator<<` qui permet d'afficher les `Expr` simpifiées et les `ConstraintBuilder`

8.2.3 Astuces techniques et exemples d'utilisation

Pour éviter les simplifications trop nombreuses, une expression n'est simplifiée que lors d'un affichage ou d'un ajout au `Model`. Ceci peut poser certains problèmes lors de l'appel de la fonction `Size`.

Il faudra donc ne pas s'étonner quand à certaines incohérences apparentes de variation de taille lors d'une trace de la taille d'une `Expr` par exemple.

```
    Model modele;

    FloatVarArray v(modele,3,0,Infinity,"v");
    FloatVar x(modele,0,10.0f,"x");
    FloatVar y(modele,0,10.0f,"y");
    FloatVar z(modele,0,10.0f,"z");

    Expr e = -3*(x + 2*y -z + 1) + 6*y - 3*z + 6;

    cout << ( Sum(v) <= e ) << endl;

//Sortie:
// v0+v1+v2<=3-3x
```

8.3 Ce qu'il faut savoir d'autre

La création des `Expr` se fera sans problème car le compilateur va être obliger de tout convertir en `Expr` lors de l'analyse des déclarations implicites des expressions linéaires. En fait, il n'y a pas d'ambiguïté connue qui pourrait venir empêcher la création d'`Expr` et c'est cela qui assure la stabilité de la manipulation d'Expr. En *C++*, il est normalement dangereux d'utiliser des constructeurs implicites, mais ici, le risque est maitrisé.

Les ConstraintBuilder ne sont pas simplifiés tant qu'ils ne sont pas intégrés à la classe `Model` par la fonction `Add`, mais généralement ils ne servent que d'objets temporaires au moment de l'appel de `Add`

9 Les classes auxiliaires *Nom*, *Nuplet* et *Omega*

9.1 Description

La classe `Nom` a pour but d'aider la création des noms des contraintes. C'est en fait un raccourci pour créer des noms composés d'un paramètre qui varie comme entier. Grâce à de nombreux constructeur et à une surcharge de flux conséquente, il est possible de créer facilement des noms indicés.

La classe `Nuplet`$< N, T >$ est en fait un tableau statique de N éléments de type T, indexés comme en $C++$ de 0 à $N-1$. Elle est utilisée pour indicer les familles de variables.

La classe `Omega`$< I, N, T^{>}$ est en fait une Propriete sur les ensembles de `Nuplet`$< N, T >$ qui est vrai lorsque l'élément d'indice I du `Nuplet`$< N, T >$ à une certaine valeur. Pratiquement, elle est utilisée par exemple pour récupérer tous les arcs (i, j) sortants d'un sommet k, c'est-à-dire l'ensemble $\Omega_k^+ = \{(k, j) \in A\}$ pour un graphe $G = (S, A)$.

9.2 Utilisation

9.2.1 Construction

On ne va pas ici détailler tous les constructeurs de `Nom`. Ils sont tous de la forme :
`Nom([string][, param][, string][, param][, string])`.
On peut aussi construire un `Nom` vide puis lui ajouter du texte et des paramètres ensuite.

La construction d'un `Nuplet` dans le code n'est pas très évidente. Il faut déclarer un `Nuplet` puis remplir le tableau qu'il contient élément par élément. L'autre possibilité offerte est la construction de Nuplet `à` partir de flux. Si l'`operator>>` a été surchargé pour l'élément T, il est possible remplir le tableau d'un `Nuplet` par l'`operator>>` pour le `Nuplet`.

La `Propriete Omega` est généralement temporaire et agit plus comme un foncteur que comme une classe. Pour construire `Omega`, il faut lui spécifier la valeur du I-ème élément comme argument de son seul constructeur. Elle n'a ni constructeur par défaut, ni constructeur de recopie.

9.2.2 Méthodes

Les méthodes de `Nom` sont les suivantes :
- `GetNom` qui permet de récupérer le nom créé
- `Clear` qui permet d'effacer le nom créé
- `SetNom` qui permet de redéfinir le nom créé
- `operator()` qui permet de récupérer le nom créé
- `operator()` qui permet de rajouter des éléments au `Nom` créé

La seule méthode de `Nuplet` est l'`operator[ ]` qui permet un élément du tableau statique.
La seule méthode de `Omega` est l'`operator()` qui permet de tester la propriété sur un `Nuplet`.

9.2.3 Astuces techniques et exemples d'utilisation

Ne pas oublier pour `Omega` que le premier nombre d'un couple à l'indice 0.

```
    cout << Nom("Contrainte",1,"_",2)() << endl;
    Nom c("Contrainte");
    c << 1 << "_" << 3;
    cout << c() << endl;

//Sortie :
// Contrainte1_2
// Contrainte1_3
```

10 Les classes *Constraint* et *LagrangianParam*

10.1 Description

La classe `Constraint` est une classe permettant de gérer les contraintes. Elle sert à gérer des contraintes sans passer par la classe `Model`, surtout concernant la relaxation. Par contre, la classe `LagrangianParam` sert exclusivement à la relaxation lagrangienne. En effet, grâce à cette classe, il est possible de paramétrer le relâchement des contraintes et ainsi appliquer des algorithmes de recherche de la meilleure borne du problème non relaché.

10.2 Utilisation

10.2.1 Construction

La classe `Constraint` se construit surtout par le retour de deux méthodes de la classe `Model` :
- `Add` qui retourne une classe `Constraint` lors de la création d'une contrainte dans le programme linéaire
- `GetConstraint` qui retourne une classe `Constraint` correspondant à ses paramètres.

La classe `LagrangianParam` va être créer soit par retour de la méthode `GetLagrangianParam` de `Constraint`. soit par les fonctions `GetInfluentLagrangianParam` ou `GetAllLagrangianParams` de la classe `Model`

10.2.2 Méthodes

Les méthodes de `Constraint` sont :
- `GetId` qui permet de récupérer l'identifiant de la contrainte,
- `GetModel` qui permet de récupérer un pointeur sur le Model auquel est rattaché la contrainte
- `Relax` qui permet de relaxer une contrainte suivant une valeur du `LagrangianParam`
- `GetLagrangianParam` qui permet la récupération du GetRelaxedConstraintExpr attaché à la contrainte
- `Rename` qui permet de renommer une contrainte
- `AddExpr` qui permet d'ajouter une `Expr` à la partie linéaire de la contrainte sans toucher à son second membre
- `GetExpr` qui permet la récupération de l'`Expr` de la partie linéaire de la contrainte avec ou sans toucher le second membre
- `GetExpr` qui permet la récupération de l'`Expr` de la partie linéaire de la contrainte relachée avec ou sans toucher le second membre
- `GetSecondMember` qui permet la récupération du second membre de la contrainte
- `SetSecondMember` qui permet la redéfinition du second membre de la contrainte
- `GetValue` qui permet de récupérer l'activité de la contrainte

Les méthodes de `LagrangianParam` sont :
- `GetId` qui permet de récupérer l'identifiant de la contrainte auquel est rattaché le multiplicateur de Lagrange.
- `GetModel` qui permet de récupérer un pointeur sur le Model auquel est rattaché le multiplicateur de Lagrange.
- `GetValue` qui permet de récupérer la valeur du multiplicateur de Lagrange.
- `SetValue` qui permet de redéfinir la valeur du multiplicateur de Lagrange.
- `operator=` qui permet de redéfinir la valeur du multiplicateur de Lagrange.
- `operator+=` qui permet d'ajouter une valeur au multiplicateur de Lagrange.
- `operator-=` qui permet de soustraire une valeur au multiplicateur de Lagrange.

10.2.3 Astuces techniques et exemples d'utilisation

Il est possible de corriger une contrainte en rajoutant un terme correctif à l'inégalité. Il est possible d'utiliser dans tous les cas la méthode AddExpr, même pour le second membre même si il est plus pratique et rapide d'utiliser SetSecondMember. Ceci est utile surtout pour modifier un programme linéaire déjà existant. Il faut penser dans ce cas à utiliser les constructeurs `NumVar(Model *, VarId)` et `Constraint(Model *, VarId)`, mais aussi les méthodes `GetMaxVarId` et `GetMaxConstraintId` de la classe `Model`.

```
   Model modele;

   FloatVar x(modele,0,Infinity,"x");
   FloatVar y(modele,0,Infinity,"y");

   modele.Minimize(2*x+3*y,"obj");
   Constraint c1 = modele.Add( x - y <= 3, "c1");
   Constraint c2 = modele.Add( 2*x + 3*y <= 3, "c2");
   c1.Relax(2.0f);

   cout << modele << endl;

//Sortie
// Minimize
// \Constante à ajouter: 6
// obj: 5y
//
// Subject to
// c2: 2x+3y<=3
//
// End
```

10.3 Ce qu'il faut savoir d'autre

Dans la relaxation lagrangienne, il est aussi possible de relâcher la nature des variables. C'est les méthodes `RelaxInt2Float` et `RelaxBool2Float` de `Model` qui permettent de le faire.

Dans le format LP, il n'y normalement pas de coefficient affine dans l'expression de la fonction objectif. Lors d'un affichage ou d'un enregistrement, la constante potentielle apparaîtra en commentaire avant la fonction objectif.

11 La classe *Model*

11.1 Description

La classe `Model` est le coeur de MODELIB. Sans elle, il est impossible de créer des variables et des contraintes. A chaque classe `Model` correspond un programme linéaire. C'est grâce à cette classe que vont être réalisée toutes les opérations sur les programmes linéaires, y compris la relaxation lagrangienne, la résolution et la création de comptes-rendus.

11.2 Utilisation

11.2.1 Construction

Il suffit de déclarer une instance de la classe `Model` pour créer un programme et pour disposer de toutes les fonctions de cette classe.

11.2.2 Méthodes

- **Construction du programme linéaire**
  - `SetProblemName` permet de définir un nom pour le problème.
  - `Maximize` permet de saisir l'expression de la fonction objectif ainsi que son nom en vue de sa maximisation.
  - `Minimize` permet de saisir l'expression de la fonction objectif ainsi que son nom en vue de sa minimisation.
  - `Add` permet d'ajouter une contrainte au modèle. La contrainte peut-être créée grâce à une classe ConstraintBuilder préalablement définie ou directement avec les opérateurs : ==, <=, >=, <, >. Il est grandement préférable de définir un nom pour chaque contrainte.

- **Gestion de la résolution**
  - `InitSolver` permet d'initialiser le solveur, c'est-à-dire choisir le solveur que l'on va utiliser, et accessoirement le nom du fichier LP temporaire où sera stocké le programme linéaire ou encore le chemin du solveur. la valeur de la fonction objectif.
  - `Solve` permet de résoudre le programme linéaire. Nécessite une initialisation préalable du solveur et retourne
  - `SetAPI` permet de définir un solveur tel que GLPK et CPLEX. et leurs valeurs.
  - `PushOptions` permet d'ajouter des options pour GLPK.
  - `RemoveOptions` permet d'enlever les options pour GLPK.

- **Gestion des informations sur le programme linéaire et sa résolution**
  - `Out` permet d'afficher le programme tel qu'il est en mémoire comme le contenu de la matrice creuse ou les variables.
  - `OutDimensions` permet d'afficher les caractéristiques du problème comme le nombre de variable, la taille de matrice creuse, etc ...
  - `OutNumVarValues` permet d'afficher la liste des variables et de leurs valeurs.
  - `OutConstraintValues` permet d'afficher la liste des contraintes et de leurs valeurs.
  - `MatriceOccupation` permet d'obtenir le pourcentage d'occupation de la matrice creuse.
  - `operator<<` permet d'afficher le programme linéaire sous le format LP.
  - `ShowMe` permet d'afficher des informations de la résolution telles que la liste des variables (non nulles ou pas)
  - `SolveInfos` permet de récupérer des informations sur la fonction objectif.
  - `SortieHMTL` permet d'obtenir un compte-rendu de la résolution sous le format HTML. Ne peut être appelée que si le format HTML est géré par la version de MODELIB
  - `SortieLATEX` permet d'obtenir un compte-rendu de la résolution sous le format LaTeX. Il faut le quand même compiler le fichier TEX généré. Ne peut être appelée que si le format HTML

est géré par la version de MODELIB
  - `ExportSparseMatrixPNG` permet d'exporter la matrice creuse vers un dessin en PNG.
  - `PrintDual` permet d'écrire le dual dans un fichier sous le format LP.
  - `PrintFraction` permet d'activer l'affichage des valeurs des variables sous forme de fraction.
  - `EndFraction` permet de désactiver l'affichage fractionnaire
  - `LoadLP` permet de charger un programme linéaire à partir d'un fichier LP.

- **Gestion des contraintes et des variables**
  - `SetNumVarValues` permet de définir les valeurs d'une liste de variables en fonction d'une liste de valeurs.
  - `SetConstraintValues` permet de définir les valeurs d'une liste de contraintes en fonction d'une liste de valeurs.
  - `EvalueConstraint` permet d'évaluer la valeur d'une contrainte en fonction de son identifiant.
  - `GetConstraint` permet de récupérer une contrainte soit par son identifiant(ordre chronologique partant de 1), soit par son nom, cette dernière méthodes étant plutôt lente.
  - `RenameConstraint` permet de renommer la contrainte dont l'identifiant est celui passé en paramètre.
  - `GetMaxVarId` permet de récupérer le plus grand identifiant qu'une variable peut avoir.
  - `GetMaxConstraintId` permet de récupérer le plus grand identifiant qu'une contrainte peut avoir.
  - `RemoveVariable` permet de supprimer une variable du programme linéaire
  - `RemoveConstraint` permet de supprimer une contrainte du programme linéaire
  - `AddToConstraint` permet d'ajouter une `Expr` à la partie linéaire
  - `SetSecondMember` permet de changer le second membre d'une contrainte.
  - `GetSecondMember` permet de récupérer le second membre d'une contrainte.
  - `Clear` permet de remettre à zéro le programme linéaire
  - `GetObjectif` qui permet de récupérer la fonction objectif

- **Gestion de la relaxation lagrangienne**
  - `IsRelaxed` permet de savoir si le problème est relaxé.
  - `SetRelaxed` permet de définir le problème comme relâché ou non. Influe sur l'affichage et la résolution.
  - `RelaxInt2Float` permet de considérer toutes les variables entiers comme des varaiables réelles.
  - `RelaxBool2Float` permet de considérer toutes les variables booléennes comme des varaiables réelles.
  - `RelaxConstraint` permet de relâcher une contrainte et l'ajouter à la fonction objectif modulo un multiplicateur de Lagrange dont une valeur peut lui être ici attribuée.
  - `Relax` a le même rôle que `RelaxConstraint` sauf que son premier paramètre est une contrainte.
  - `GetInfluentLagrangianParam` permet de récupérer la liste des multiplicateurs de Lagrange qui violent leur contrainte associée et celle des multiplicateurs des contraintes non saturées.
  - `GetAllLagrangianParams` récupére un vecteur de tous les mutliplicateurs de Lagrange.
  - `UnrelaxModel` permet de revenir au problème non relâché, y compris au niveau des variables entières et booléennes.

- **Gestion du dual**
  - `Normalize` permet de normaliser le modèle, c'est-à-dire de mettre les contraintes sous la forme d'une matrice A tel que $Ax \leq b$ où x est le vecteur inconnu et b le second membre. Cette fonction sert à obtenir un dual correct.
  - `GetDual` permet d'obtenir le programme linéaire correspondant au dual du programme dans Model.

11.2.3 Astuces techniques et exemples d'utilisation

Pour sauver un fichier LP, il suffit d'utiliser la surcharge de flux dans un fichier ouvert par une classe `ofstream`.

```
//Une classe Model modele a été construite au préalable avec un programme →
    ↪ linéaire

ofstream file("monFichier.lp");  //création du flux

file << modele;      //enregistrement dans le fichier monFichier.lp

file.close();        //vider le tampon d'écriture
```

11.3 Ce qu'il faut savoir d'autre

Une variable détruite est enlevée de toutes les contraintes. Cette opération et lente et couteuse. De manière générale, les fonctions qui modifient la matrice creuse sont lentes et leurs utilisations doivent être faites avec parcimonie.

Deuxième partie

Exemples de code

12 Un premier programme linéaire

Ce programme est un exemple pour se familiariser avec la création de variable, la déclaration de la fonction objectif et l'ajout de contrainte

```
#include <Modelib.h>
using namespace std;

int main( int argc, char ** argv) {
  //-------------------------------------------------------
  //  Déclaration des variables
  //-------------------------------------------------------

  // déclaration de la classe Model qui va représenté le programme linéaire
  Model modele;

  //déclaration des variable dans le programme linéaire
  FloatVar x(modele,0,30.23,"prems"); // déclare une variable réelle
  cout << "Le nom de la variable x est:" << x.GetName() << endl;

  /* manière équivalente:
  FloatVar x(modele);
  x.SetName("prems");
  x.SetLowerBound(0);
  x.SetUpperBound(30);  */

  // déclaration d'une variable entiere
  IntVar iVar(modele,0,17,"i");
  // déclaration d'une variable booléene
  BoolVar bVar(modele,"b");

  // déclaration d'un vecteur de valeur:
  IntArray iVect(3,2);
  // déclaration d'un vecteur de variable:
  IntVarArray iVectVar(modele,3,0,50,"i");

  // déclaration d'une expression
  Expr e = iVect*iVectVar - Sum(iVectVar) + 3*(bVar - 4*iVar);

  //-----------------------------------------------------
  //  Manipulation du programme linéaire
  //-----------------------------------------------------

  //définition de la fonction objectif
   modele.Minimize( iVar + bVar*4 , "obj" );

   /* manière équivalente:  modele.Maximize( -4*bVar - iVar, "obj");*/

   // On ajoute une contrainte d'égalité
   modele.Add( bVar == 1, "C1" );

   //On ajoute une contrainte d'infériorité
   modele.Add( e <= 20, Nom("C",2));

   //on ajoute une contrainte de supériorité
   modele.Add( iVar <= iVectVar[0] + 2*iVectVar[1] + 3*iVectVar[2], "C3" );

   cout << modele << endl;

   return 0;
}
```

12.1 Résultats d'exécution

```
Le nom de la variable x est:prems
\
Minimize obj: i+4b

Subject to
C1: b=1
C2: -12i+3b+i0+i1+i2<=20
C3: i-i0-2i1-3i2<=0

Bounds
prems<=30.23
i<=17
i0<=50
i1<=50
i2<=50

Generals
i
i0
i1
i2

Binaries
b

End
```

12.2 Commentaire

La première des choses à faire est de déclarer le conteneur du programme linéaire en instanciant une classe `Model`. On peut ensuite déclarer les variables du programme linéaire en leur donnant un nom et des bornes. On peut donner n'importe quel nom à une variable, même un nom déjà donné, mais il est conseillé de nommer les variables à l'identique dans le code *C++* et dans le programme linéaire.

On déclare ensuite un tableau de variable. L'opération qui suit est inutile mais montre juste les possibilités des expressions, à savoir que toutes les opérations linéaires de base peuvent être faites ainsi que des produits scalaire. Mais l'utilité des expressions se verra plus tard.

Vient ensuite l'écriture du programme linéaire. On peut définir la fonction objectif facilement et à n'importe quel moment du programme (avant la résolution tout de même). On peut aussi la redéfinir plus tard dans le programme(si un algorithme le demande). Lui donner un nom ajoute une petite touche de personnalisation mais n'est pas indispensable.

On peut ensuite définir le coeur du programme linéaire : les contraintes. Par simple appel de la fonction Add, on définit une contrainte, avec une expression, un opérateur, et un scalaire ou une expression. Dans le cas de deux expressions, on peut constater en regardant les résultats que la fonction a été mise sous une forme standard (simplifiée sous une forme matricielle). Il y plusieurs types d'opérateurs, mais habituellement on n'utilise que ==, <= et >=. Attention néanmoins à ne pas confondre == et =.

Comme résultats, on peut voir le programme sous le format LP. C'est ainsi qu'il va être exploité par les solveurs. Se reporter à une documentation sur les fichiers LP pour en savoir plus.

13 Exemple de résolution

Ce code C++ permet de réaliser la création d'un problème linéaire, sa résolution et les opérations d'entrée/sortie.

```
#include <Modelib.h>

using namespace std;

int main( int argc, char ** argv)
{
  /*------------------------------------------------------------------------*/
  /*  Exemple d'un restaurateur:                                            */
  /*   Un restaurateur constate que sa clientèle préfère                    */
  /*   -des assiettes à 8 euros, contenant 5 oursins, 2 bulots et 1 huître  */
  /*   -des assiettes à 6 euros, contenant 3 oursins, 3 bulots et 3 huîtres.*/
  /*    Il dispose de 30 oursins, 24 bulots et 18 huîtres.                  */
  /*                                                                        */
  /*------------------------------------------------------------------------*/

  //-------------------------------------------------------
  //  Déclaration des variables
  //-------------------------------------------------------

  // déclaration de la classe Model qui va représenté le programme linéaire
  Model modele;

  //déclaration des variable dans le programme linéaire
  IntVar x1(modele,"x1"); // nombre d'assiettes à 8 euros
  IntVar x2(modele,"x2"); // nombre d'assiettes à 6 euros

  modele.SetProblemName("Probleme du restaurateur");
  modele.Maximize( 8*x1 +  6*x2, "z" );

  modele.AddComment("Nombre d'oursins");
  modele.Add( 5*x1 + 3*x2 <= 30, "c1");
  modele.AddComment("Nombre de bulots");
  modele.Add( 2*x1 + 3*x2 <= 24, "c2");
  modele.AddComment("Nombre d'huitres");
  modele.Add(   x1 + 3*x2 <= 30, "c3");

  modele.InitSolver(GLPK);
  float solution = modele.Solve();

  cout << endl << "Valeur de la solution: " << solution << endl;

  modele.ShowMe();

  #ifdef USE_HTML_FORMAT
    modele.SortieHTML("Restaurateur.html");
  #endif
  #ifdef USE_LATEX_FORMAT
    modele.SortieLATEX("Restaurateur.tex");
  #endif
}
```

13.1 Résultats d'execution

```
\Probleme du restaurateur
```

```
Maximize
z: 8x1+6x2

Subject to
\Nombre d'oursins
c1: 5x1+3x2<=30
\Nombre de bulots
c2: 2x1+3x2<=24
\Nombre d'huitres
c3: x1+3x2<=30

Generals
x1
x2


End

lpt_read_prob: reading LP data from `./tmp.solver.lptmp.lp'...
lpt_read_prob: 2 variables, 3 constraints
lpt_read_prob: 18 lines were read
lpx_simplex: original LP has 3 rows, 2 columns, 6 non-zeros
lpx_simplex: presolved LP has 3 rows, 2 columns, 6 non-zeros
lpx_adv_basis: size of triangular part = 3
*     0:   objval =  0.000000000e+000   infeas =  0.000000000e+000 (0)
*     2:   objval =  5.600000000e+001   infeas =  0.000000000e+000 (0)
OPTIMAL SOLUTION FOUND
Integer optimization begins...
Objective function is integral
+     2: mip =     not found yet; lp =  5.600000000e+001 (1, 0)
+     4: mip =  5.400000000e+001; lp =  5.600000000e+001 (3, 2)
+     6: mip =  5.400000000e+001; lp =     tree is empty (0, 5)
INTEGER OPTIMAL SOLUTION FOUND
Time used:   0.0 secs
Memory used: 0.2M (157548 bytes)
lpx_print_mip: writing MIP problem solution to `./tmp.solver.lptmp.lp.sol'...

Valeur de la solution: 54
Valeurs des variables (type et domaine de définition):
x1 = 3  ........ INT ....... [0,+Infinity]
x2 = 5  ........ INT ....... [0,+Infinity]
Caracteristiques des constraintes (si non toutes saturées):
```

13.2 Commentaire

On constatera que le programme linéaire se construit de la même façon que précédemment mais qu'il plus simple, donc plus facile à mettre en place (la description de se programme est en commentaire au début du programme).

Les plus de la construction du programme sont la définition du nom du problème et les commentaires. Ils sont utiles pour savoir de quoi un programme traite, notamment lors d'un archivage et séparer les contraintes en blocs.

Le plus important vient ensuite : la résolution du problème. Pour ce faire, il faut déjà initialiser le solveur. Son oubli se traduira par une erreur d'exécution caractéristique. Puis il suffira d'appeler simplement la fonction `Solve` de la classe `Model`. Celle-ci renvoie la fonction de la valeur objectif. Pour voir l'état des variables il faut appeler la fonction `ShowMe`.

Les résultats d'exécution sont composés de l'affichage du programme, de la sortie texte de GLPK, et de l'affichage des résultats de la résolution.

Et voila un premier programme résolu simplement avec Modelib.

Dans le code vient ensuite un appel de `SortieHTML` et de `SortieLATEX` qui écrivent dans les fichiers spécifiés un petit compte-rendu de la résolution. Les `#ifdef` viennent des options d'installation de la librairie. Dans le cas où la librairie a été compilé avec l'option HTML ou l'option LATEX, les fonctions ne sont pas accessibles.

Les fichiers de compte rendu sont disponibles normalement avec ce tutoriel. Voici quand même un petit aperçu de la sortie HTML :

14 Exemple d'utilisation des NumVarFamily

Ce programme met en oeuvre un outil qui n'est pas aisé a mettre en place : les NumVarFamily.

```
#include <Modelib.h>
using namespace std;

int main( int argc, char ** argv ) {
  //----------------------------------------
  // Declarations et saisie des paramètres
  //----------------------------------------
  Model modele;

  unsigned nbSommets;
  cout << "Nombre de sommets" << endl;
  cin >> nbSommets;

  unsigned origine;
  unsigned destination;
  float flux;

  cout << "Sommets de depart:" << endl;
  cin >> origine;
  origine = origine%nbSommets;
  cout << "Sommets d'arrivee:" << endl;
  cin >> destination;
  destination = destination%nbSommets;
  cout << "Flux devant circuler entre les deux" << endl;
  cin >> flux;
  flux = ( flux < 0? -flux:flux )

  list<Couple> tmpList;
  Ensemble<float> couts;

  //------------------------------------------------
  // Construction du graphe inutile
  //------------------------------------------------
  Couple tmpCouple;
  for(unsigned i = 0; i < nbSommets; ++i)
    for(unsigned j = i + 1; j < nbSommets; ++j )
    {
        tmpCouple[0] = i;
        tmpCouple[1] = j;
      tmpList.push_back(tmpCouple);
      couts.Add( (19.0*i+17.0*j)/36.0 );
    }

  tmpCouple[0] = nbSommets-1;
  tmpCouple[1] = 0;
  tmpList.push_back(tmpCouple);
  couts.Add( 0.0 );

  //On recupere le nombre d'arcs
  unsigned nbArcs = tmpList.size();
```

```
  //on cree la NumVarFamily
  Ensemble<Couple> arcs(tmpList);
  CoupleFloatVarFamily y(modele,0,flux,arcs,"y_%_%");

  modele.SetProblemName("Probleme du graphe inutile");

  //La fonction objectif est classiquement la sommme des (couts*flux sur l'arc)
  Expr objectif;
  for(unsigned i =0; i<nbArcs; ++i)
      objectif+= couts.Get(i)*y.Get(i);

  modele.Maximize(objectif,"objectif");

  //On contruit les contraintes de conservation de flux
  for(unsigned j=0;j<nbSommets;++j) //pour chaque sommets
  {
      //On prend la partie de l'ensemble des variables représentant la quantité
      //de flux sur les arcs sortant du sommet j
      NumVarArray t = y.GetFamilyPart( Omega<0,2,unsigned>(j) );
      //On prend la partie de l'ensemble des variables représentant la quantité
      //de flux sur les arcs entrant dans le sommet j
      NumVarArray u = y.GetFamilyPart( Omega<1,2,unsigned>(j) );
      //L'origine et la destination sont des cas particuliers
      if( j != origine && j != destination )
      {
          //Le flux est conservé, ie la quantité de flux rentrante
          //est égale à la quantité de flux sortante
          modele.Add( Sum(t) - Sum(u) == 0, Nom("c1_",j) );
      }
      else
      {
          if( j == origine )
          {
            //La quantité de flux sortant de l'origine est la quantité de flux
            modele.Add( Sum(t) == flux, Nom("c1_",j) );
            //La quantité de flux entrant dans l'origine dans est nulle
            modele.Add( Sum(u) == 0, Nom("c1_") << j << "_2" );
          }
          else
          {
            modele.Add( Sum(t) == 0, Nom("c1_",j) );
            //La quantité de flux sortant de la destination est nulle
            modele.Add( Sum(u) == flux, Nom("c1_") << j << "_2" );
            //La quantité de flux entrant dans la destination est la quantité →
                ↪ de flux
           }
      }
  }

  cout << "---------Resolution--------" << endl;
  modele.InitSolver(GLPK); //ne pas oublier
  float solution = modele.Solve();

  cout << endl << "La solution est de: " << solution << endl;
  modele.ShowMe();
  #ifdef USE_HTML_FORMAT
    modele.SortieHTML("GrapheInutile.html");
  #endif

  return 0;
}
```

14.1 Résultats d'execution

```
Nombre de sommets
5
Sommets de depart:
3
Sommets d'arrivee:
1
Flux devant circuler entre les deux
1.0
---------Resolution--------
lpt_read_prob: reading LP data from `./tmp.solver.lptmp.lp'...
lpt_read_prob: 11 variables, 7 constraints
lpt_read_prob: 28 lines were read
lpx_simplex: original LP has 7 rows, 11 columns, 22 non-zeros
Objective value = 3.944442
OPTIMAL SOLUTION FOUND BY LP PRESOLVER
Time used:   0.0 secs
Memory used: 0.1M (66492 bytes)
lpx_print_sol: writing LP problem solution to `./tmp.solver.lptmp.lp.sol'...

La solution est de: 3.94444
Valeurs des variables (type et domaine de définition):
y_0_1 = 1  ........ FLOAT ....... [0,1]
y_3_4 = 1  ........ FLOAT ....... [0,1]
y_4_0 = 1  ........ FLOAT ....... [0,1]
Caracteristiques des constraintes (si non toutes saturées):
```

14.2 Commentaire

Cet exemple n'est absolument pas relié à un vrai problème de recherche opérationnelle. Il permet néanmoins de manipuler les outils de base rattachés au NumVarFamily. Il ne traite pas du problème par exemple de la lecture de données dans des fichiers ou la création de Propriete. Il est déjà suffisamment compliqué bien qu'il n'utilise pas les fonctionnalités les plus avancées de MODELIB.

Le code commence par la déclaration du programme linéaire puis par la saisie des données intrinsèques du problème. Puis vient la déclaration d'une liste de `Couple` et d'un Ensemble de `float`. Surtout pas de panique : Couple n'est rien d'autre qu'une redéfinition de `Nuplet<2,unsigned>`. On se rend compte tout de suite qu'on va devoir manipuler des objets "templates". Il est nécessaire de savoir manipuler ceux-ci avant de continuer. La liste de `Couple` va permettre de construire un ensemble d'arcs qui vont définir un graphe dont les sommets sont numérotés de `0` à `nbSommets`. L'ensemble des coûts des arcs est représenté par un ensemble, mais il permet pas de dégager tout le potentiel de cette structure. Il faut voir que l'on aurait pu utiliser n'importe quel autre conteneur sans avoir besoin de changer fondamentalement le code.

Une liste de Couple est donc crée puis remplie grâce à un objet intermédiaire. Une question pertinente serait de demander s'il n'y a pas de constructeur de Couple qui permettent d'ajouter directement les données à la liste. Cette absence provient du caractère générique de `Nuplet`, des problèmes de typename liés aux classes templates, et du fait que le constructeur de Couple soit trop spécifique.

Ensuite vient la construction de la NumVarFamily. Elle s'effectue en deux étapes :

- la définition de l'Ensemble des index à partir de la liste précédemment créée
- la Construction de la NumVarFamily (`CoupleFloatVarFamily` n'est qu'une redéfinition de `FloatVarFamily<2,unsigned>`) qui sera de la forme y_i_j où les couples (i, j) appartiennent à l'`Ensemble arcs`.

Il faut ici bien comprendre ce qu'est une `NumVarFamily`. C'est un ensemble de variable qui sont indicées par un autre ensemble. Par exemple, on peut voir un `NumVarArray` comme une `NumVarFamily<1,unsigned>`

avec comme ensemble d'indices la liste des entiers de 0 à la dimension du tableau moins un. De même on peut voir une `NumVarMatrix` comme une `NumVarFamily<2,unsigned>` avec comme ensemble d'indices la liste des couples d'entiers du produit cartésien de deux listes d'entiers de 0 à la dimension du tableau moins un.

Le gros avantage des NumVarFamily est que l'ensemble d'entiers peut être n'importe quel ensemble, y compris un sous-ensemble de N entiers non contigus. Il y a certes des contraintes sur les ensembles d'indices, ils doivent posséder une surcharge certains opérateurs tels `<<` ou encore `>>`, et leur affichage doit permettre la formation d'un nom de variable. Il est conseillé, avant d'utiliser les `NumVarFamily` avec d'autre ensemble que `Nuplet<N,unsigned>`, de bien connaître la généricité et de regarder comment a été codée la classe NumVarFamily dans le cas de problèmes de compilation. Pour tout problème ou question, ne pas hésiter à envoyer des courriels aux auteurs de MODELIB. De toute façon, les `NumVarFamily` sont forcément basées sur des Nuplets. Seul le type de ceux-ci et leur dimension vont faire varier les types de `NumVarFamily`.

La famille de variables est donc ici indicées sur un ensemble d'arcs d'un graphe orienté et représente un flux. Cela veut dire qu'à chaque arcs du graphe, il y a une variable dans le programme linéaire qui servira à déterminer la quantité de flux qui devra circuler sur cette arête. S'il y avait plusieurs commodités, il y aurait plusieurs variables. Dans ce cas il est conseillé d'utiliser un tableau de NumVarFamily plutôt qu'une NumVarFamily de dimension supérieure pour des problèmes de rapidité essentiellement.(Bien codée, la deuxième solution n'est pas si lente par rapport à la première solution, il faut pour cela définir des parties représentant les commodités.)

Ensuite vient la définition de la fonction objectif. Celle-ci est la somme des produits de la quantité de flux sur chaque arcs par les coûts correspondants, c'est-à-dire $\sum_{(i,j)\in arcs} c_{ij}y_{ij}$. Pour construire l'expression de cette fonction, il faut prendre tous les éléments de la famille et les multiplier par leur coûts respectifs. Et ici, on peut se poser une deuxième question pertinente : comment associer les coûts aux variables ? Il faut pouvoir identifier aussi bien les coûts que les variables, et qui plus est, avec des systèmes compatibles. La solution adoptée a été d'identifier les objets, aussi bien les variables que les coûts, par leur ordre de création codé par un entier non signé. De cette manière, il suffit de définir en même temps un coût et une variable pour qu'ils aient le même identifiant. Le plus grand identifiant qui est définit par la taille d'un ensemble moins un. Dans l'exemple, pour chaque arcs que l'on crée, on ajoute en même temps un coût à l'ensemble des coûts de telle sorte qu'ils correspondent au niveau de leurs identifiants. D'où les deux appels aux fonctions `Get` avec le même identifiant en paramètre.

Une fois la fonction objectif définie, on définit les contraintes de conservation de flux. Pour chaque sommet, il faut que le flux rentrant soit égal au flux sortant. Il faut donc récupérer les variables de flux dont les arcs ont ce sommet comme extrémité. C'est le rôle des `Propriete Omega`. Ces propriétés sont paramétrables, encore une fois grâce à la généricité. `Omega<I,N,T>(x)` permet d'extraire d'un ensemble de `Nuplet<N,T>` le sous-ensemble des nuplets tels que le I^{eme} élément de chaque nuplet vaille x (Attention pour le premier élément, I vaut 0). On peut ainsi récupérer les arcs sortant d'un sommet j grâce à `Omega<0,2,unsigned>(j)` et les arcs rentrant d'un sommet j grâce à `Omega<1,2,unsigned>(j)`. On va appeler la fonction GetFamilyPart pour récupérer un vecteur contenant toutes les variables vérifiant une des propriétés précédentes et on va faire la somme de ses termes pour avoir d'une part la quantité de flux sortante et d'autre part la quantité de flux rentrante. Les cas exceptionnels sont les sommets de destination et d'origine qui nécessite un traitement particulier.

Il ne reste plus qu'initialiser le solveur, résoudre et regarder le compte-rendu HTML ou LaTeX.

15 Exemple de relaxation lagrangienne

Un petit exemple qui permet de voir comment implémenter un algorithme de relaxation lagrangienne avec MODELIB

```
#include <Modelib.h>

using namespace std;

int main( int argc, char ** argv) {
    // declaration du modele
    Model knap;

    // Init du solver
    knap.InitSolver(GLPK); //initialisation du solveur

    //Declarations des variables
    BoolVar x1(knap,"x1");
    BoolVar x2(knap,"x2");
    BoolVar x3(knap,"x3");
    BoolVar x4(knap,"x4");

    //-------------------------------------------------------------
    //Creation du probleme lineaire: knapsack legerement modifie
    //-------------------------------------------------------------
    knap.SetProblemName("Relaxation lagrangienne sur un probleme de sac-à-dos") →
        ↪ ;
    //Fonction objectif
    knap.Maximize( 4*x1 + 5*x2 + 6*x3 + 7*x4  );

    knap.Add( 4*x1 + 5*x2 + 6*x3 + 7*x4 >= 0, "cprglpk");
    //Contrainte qui permet l'acceptation du probleme par cprglpk
    Constraint c1 = knap.Add( 2*x1 + 2*x2 + 3*x3 + 4*x4 <= 7, "c1");
    Constraint c2 = knap.Add( x1 - x2 + x3 - x4 <= 0 , "c2" );

    c1.Relax(0.0); //on relaxe la premiere contrainte
    c2.Relax(0.0); //on relaxe la deuxieme contrainte

    float step = 0.5;              //pas de deplacement
    float bestSol = knap.Solve(); //on initialise la meilleur borne avec le →
        ↪ vecteur lambda nul
    unsigned stagne = 0;           //nombre d'etape ou l'on trouve la meme borne
    unsigned mstagne = 3;          //le nombre maximum d'etape ou on trouve la →
        ↪ meme borne
    unsigned n = 0;                //le nombre d'iteration

    //-------------------------------------------------------------
    //boucle principale
    //-------------------------------------------------------------
    cerr << "format de sortie" << endl
         << "iteration: valeurs des multiplicateurs -> solution" << endl;
    while( step >= 1e-6 &&  n<100) //tant que le pas est superieur à  une →
        ↪ certaine precision
    {
        float nouvSol = knap.Solve();  //On calcule la nouvelle solution
        if( nouvSol < bestSol )     //On actualise la meilleure solution
        {
            bestSol = nouvSol;
            stagne = 0;
            cerr << n  <<  ":  ";
            vector<LagrangianParam> v = knap.GetAllLagrangianParams();
            for(unsigned i = 0; i < v.size(); ++i)
                cerr << v[i].GetValue() << " ";
```

```
            cerr << " -> " << nouvSol << endl;
        }
        else //sinon on stagne
            ++stagne;

        list<LagrangianParam> listeViol;    //liste des contraintes violees
        list<LagrangianParam> listeSlack;   //liste des contraintes non →
            ↪ saturees

        knap.GetInfluentLagrangianParam(listeViol,listeSlack); //on recupere →
            ↪ ces listes

        list<LagrangianParam>::iterator i = listeViol.begin();
        for( ; i != listeViol.end(); ++i)   //pour chaque constrainte violee, →
            ↪ on augmente le multiplicateur
        {
            *i += step;
        }

        if( listeSlack.size() ) //si des contraintes sont laches
        {
            list<LagrangianParam>::iterator i = listeSlack.begin();
            for( ; i != listeSlack.end(); ++i) //pour chaque constrainte lache →
                ↪ , on diminue le multiplicateur
            {
                if( i->GetValue() - step > 0)  //on verifie si le lagrangien →
                    ↪ est bien positif
                  *i -=  step;
                else
                  *i = 0;
            }
        }

        if( stagne > mstagne) //si on stagne, on diminue le pas
        {
            step*=0.5;
            stagne = 0;
        }
        ++n;
    }
    cerr << "La meilleure borne de la solution optimale est donc: " << bestSol →
        ↪ << endl;
    knap.OutNumVarValues(false,cerr);
    cerr << "La valeur de la fonction obectif pour la solution trouvée est: " →
        ↪  << knap.GetObjectif().Evalue( false ) << endl;
    knap.UnrelaxModel();
    cerr << "La solution optimale est: " << knap.Solve();
    return 1;
}
```

15.1 Résultats d'execution

```
#exemple4.exe > sortielagrange.txt
format de sortie
iteration: valeurs des multiplicateurs -> solution
1:  0 0.5 0  -> 20
2:  0 1 0  -> 18
3:  0 1.5 0  -> 16
4:  0 2 0  -> 15
15:  0 1.875 0.125  -> 14.875
```

```
16:  0 1.875 0.25  -> 14.75
26:  0 1.84375 0.3125  -> 14.6875
36:  0 1.83594 0.328125  -> 14.6719
42:  0 1.83203 0.328125  -> 14.6719
48:  0 1.83398 0.332031  -> 14.668
58:  0 1.8335 0.333008  -> 14.667
64:  0 1.83325 0.333008  -> 14.667
70:  0 1.83337 0.333252  -> 14.6667
78:  0 1.83337 0.333313  -> 14.6667
84:  0 1.83334 0.333313  -> 14.6667
90:  0 1.83333 0.333313  -> 14.6667
91:  0 1.83333 0.333328  -> 14.6667
97:  0 1.83334 0.333328  -> 14.6667
La meilleure borne de la solution optimale est donc: 14.6667
Valeurs des variables (type et domaine de définition):
x1 = 0  ........ BOOL ....... [0,1]
x2 = 1  ........ BOOL ....... [0,1]
x3 = 1  ........ BOOL ....... [0,1]
x4 = 0  ........ BOOL ....... [0,1]
La valeur de la fonction objectif pour la solution trouvée est: 13
La solution optimale est: 13
```

15.2 Commentaire

Ce code est une mise en pratique d'une heuristique de relaxation lagrangienne. On voit ici que l'on résout une centaine de programme linéaire afin de chercher une borne à la solution optimale et les coefficients de Lagrange associé. La convergence dépend évidement des paramètres choisis et l'algorithme peut être sûrement amélioré, corrigé, ou rejeté.

L'intérêt de ce code est de montrer comment gérer les multiplicateurs de Lagrange d'un programme linéaire modélisé sous MODELIB. Chaque ligne de la matrice des contraintes est multipliable par un coefficient de Lagrange et la soustraction à la fonction objectif se fait automatiquement.

Il faut tout d'abord revenir sur la relaxation lagrangienne d'un problème linéaire. Relaxer un programme linéaire, c'est enlever les contraintes difficiles d'un problème afin de faciliter sa résolution. Pour plus d'informations sur la théorie, se reporter à un cours sur la relaxation lagrangienne. Il y a plusieurs cas de figure. Ici on enlève une contrainte de l'ensemble des contraintes.

Le problème original est :

$$\begin{array}{lrcl} \text{Maximize} & 4x1 + 5x2 + 6x3 + 7x4 & & \\ \text{Subject to} & & & \\ (\times\lambda_1) & 2x1 + 2x2 + 3x3 + 4x4 & \leq & 7 \\ (\times\lambda_2) & x1 - x2 + x3 - x4 & \leq & 0 \\ & x_i \in \{0,1\} & & \end{array}$$

On multiplie la première contrainte par λ_1 et la deuxième contrainte par λ_2. Puis on les retire du programme par en les soustrayant à la fonction objectif pour garder la cohérence du programme :

$$\begin{array}{lr} \text{Maximize} & 4x1 + 5x2 + 6x3 + 7x4 \\ & -\lambda_1(2x1 + 2x2 + 3x3 + 4x4 - 7) \\ & -\lambda_2(x1 - x2 + x3 - x4) \\ \text{Subject to} & \\ & x_i \in \{0,1\} \end{array}$$

On va ensuite chercher à déterminer les meilleurs valeurs de pour λ_1 et λ_2. Pour cela on va utiliser une heuristique. Sachant que la fonction objectif est convexe par rapport à (λ_1, λ_2), on va simplement faire un recherche de proche en proche du minimum de cette fonction suivant l'algorithme :

```
Initialiser chaque multiplicateur à 0 et le pas k en fonction du problème

Tant que k > epsilon faire
   1. Résoudre le programme linéaire pour obtenir une solution
   2. Pour chaque contrainte violée, ajouter le pas k
        au multiplicateur correspondant
   3. Pour chaque contrainte non saturée, diminuer du pas k
        le multiplicateur correspondant
   4. Si m itérations se sont passées sans que la meilleure valeur
        de la fonction objectif ait diminuée, diviser le pas par 2
```

Il faut maintenant pouvoir implémenter cet algorithme. On déclare donc le programme linéaire de façon habituelle sauf que l'on récupère les classes `Constraint` retournées par la fonction `Add` de `Model`. La contrainte supplémentaire par rapport au problème initial vient du fait que GLPK requiert que le problème ait au moins une contrainte pour lancer une résolution. Il suffit donc de prendre une contrainte trivial qui n'aura pas d'influence sur le problème, comme la positivité de la fonction objectif. Il faut ensuite spécifier les contraintes que l'on veut relaxer. Pour cela, on appelle la fonction `Relax` des classes de contraintes que l'on a récupérées. Celle-ci permet de déclarer un multiplicateur de Lagrange à l'intérieur du programme linéaire et de l'initialiser.

Maintenant que les contraintes sont relaxées, il ne reste plus qu'à suivre l'algorithme pas à pas. On commence donc par initialiser des variables qui vont servir dans l'algorithme, puis on créer une boucle comme préconise l'algorithme. On résout donc le programme linéaire. On regarde si on améliore la solution, sinon on considère qu'on stagne. Ensuite il faut savoir quelles sont les contraintes violées et les contraintes non saturées. Pour cela, la fonction `GetInfluentLagrangianParam` permet de récupérer deux listes correspondant aux multiplicateurs de Lagrange des contraintes qui sont violée ou non saturée. Ensuite, on parcourt les listes afin d'appliquer l'algorithme. Une remarque concernant le test de positivité pour les contraintes non saturées. Normalement la valeur du multiplicateur ne doit pas être négative pour des contraintes de type $\leq$, d'où le test. La classe `LagrangianParam` gère cette spécificité et empêche d'attribuer une valeur négative à ces contraintes tout en affichant un avertissement. Mais afin d'éviter de remplir de warning les résultats de l'exécution, un test est effectué. Ensuite on teste si on stagne, et dans ce cas on divise le pas par deux. L'algorithme a été implémenté.

Troisième partie

Toutes les fonctions

–

```
Partie<T>:
    Partie _inter(const Partie & partie) const
    Partie _union(const Partie & partie) const
    Partie _compl(const Partie & partie) const
```

A

```
    void AddExpr(const Expr &)

Ensemble<T>:
    void Add( const T & element)
    Partie all()

Expr:
    Expr & ASS( const Num & num, const NumVar & numVar )
    Expr & ASS( const Num & num, const NumVar & numVar, const LagrangianParam →
        ↪  & mult )

Model:
    Constraint Add( const ConstraintBuilder &, const std::string & nom = "")
    Constraint Add( const ConstraintBuilder & constraint, const Nom & _nom)
    std::string AddComment( const std::string & comment )
    void AddToConstraint(VarId,const Expr &)
```

B

```
Bool:
    Bool(bool booleen = false)

BoolVar:
    BoolVar( Model &_modele, const std::string & nom = "unknow" )

BoolVarArray:
    BoolVarArray( Model &_modele, unsigned size=0, const std::string & nom = " →
        ↪ unknow", const std::string & fin = "")

BoolVarFamily<N,T>:
    BoolVarFamily( Model &_modele, const Ensemble< Nuplet<N,T> > & ensemble,  →
        ↪ const std::string & nom )

BoolVarMatrix:
    BoolVarMatrix( Model &_modele, unsigned sizeI=0, unsigned sizeJ=0, const  →
        ↪ std::string & nom = "unknow", const std::string & milieu = "", const →
        ↪  std::string & fin = "" )
```

C

```
Constraint:
    Constraint(Model & _m,VarId _id)
    Constraint( const Model & _m,VarId _id)

ConstraintBuilder:
    ConstraintBuilder( const Expr & , ConstraintBuilder::Operateur , const Expr →
        ↪ &)

Nom:
    void Clear()

Model:
    void Clear()
```

E

```
Ensemble<T>:
    Ensemble( std::list<T> base)

Expr:
    float Evalue(bool avecLagrangianParam = true) const
    Expr( const NumVar & numVar)
    Expr( float )
    Expr( double )
    Expr( int )
    Expr( unsigned )
    Expr( const Num & num)
    Expr( const Num & num, const NumVar & numVar)
    Expr( const Num & num, const NumVar & numVar, const LagrangianParam & mult)

Model:
    void EndFraction()
    float EvalueConstraint( VarId id ) const
    void ExportSparseMatrixPNG( const std::string& _nom="./MatriceCreuse.png" ) →
        ↪ const
```

F

```
Float:
    Float( float flottant = 0.0f )

FloatVar:
    FloatVar( Model &_modele, float lowerBound = 0, float upperBound = Infinity →
        ↪ , const std::string & nom = "unknow" )
    FloatVar( Model & _modele, const std::string & _nom = "unknow" )

FloatVarArray:
    FloatVarArray( Model &_modele, unsigned size=0, float _lb= 0.0f, float _ub →
        ↪ = Infinity, const std::string & nom = "unknow", const std::string &  →
        ↪ fin = "")
```

```
FloatVarFamily<N,T>:
    FloatVarFamily( Model &_modele, float _lb, float _ub, const Ensemble< →
        ↪ Nuplet<N,T> > & ensemble, const std::string & nom )

FloatVarMatrix:
    FloatVarMatrix( Model &_modele,unsigned sizeI=0, unsigned sizeJ=0,float _lb →
        ↪ = 0.0f, float _ub= Infinity,const std::string & nom = "unknow",const →
        ↪  std::string & milieu = "", const std::string & fin = "" )
```

G

```
Constraint:
    Expr GetExpr(bool secondMember = false) const
    VarId GetId() const
    LagrangianParam GetLagrangianParam() const
    Model * GetModel() const
    Expr GetRelaxedExpr(bool secondMember = false) const
    float GetSecondMember() const;
    float GetValue() const

Ensemble<T>:
    T Get(unsigned i) const
    T & Get(unsigned i)

Expr:
    float GetConstant(bool avecLagrangianParam = true) const

LagrangianParam:
    VarId GetId() const
    Model * GetModel()
    float GetValue() const

Model:
    std::vector<LagrangianParam> GetAllLagrangianParams()
    std::string GetComment( VarId id ) const
    Constraint GetConstraint(VarId id) const
    Constraint GetConstraint(const std::string &)
    Model GetDual() const
    void GetInfluentLagrangianParam( std::list<LagrangianParam> & listViolated →
        ↪ , std::list<LagrangianParam> & listSlacked)
    VarId GetMaxConstraintId() const
    VarId GetMaxVarId() const
    Expr GetObjectif() const
    float GetSecondMember(VarId) const

Nom:
    std::string GetNom() const

Num:
    MuteVar::VarType GetType() const { return type; }
    float GetValue() const {return value;}

NumArray:
    MuteVar::VarType GetType() const { return type;}

NumVar:
    float GetLowerBound() const
    Model * GetModel() const
    std::string GetName() const
    MuteVar::VarType GetType() const
```

```
    float GetUpperBound() const
    float GetValue() const
    VarId GetVarId() const

NumVarArray:
      NumVar * Get(unsigned n) const
    MuteVar::VarType GetType() const

NumVarFamily<N,T>:
    NumVar Get(unsigned i) const { return variables[i];}
    std::vector<NumVar> GetFamilyPart( const Propriete< Nuplet<N,T> > &  →
       ↪ propriete)
    std::vector<NumVar> GetFamilyPart( const typename Ensemble< Nuplet_T >:: →
       ↪ Partie & p)

NumVarMatrix:
    NumVar * Get(unsigned i, unsigned j);
    NumVarArray GetCol(unsigned j) const
    unsigned GetNbCols() const
    NumVarArray GetRow(unsigned i) const
    unsigned GetNbRows() const
    MuteVar::VarType GetType() const { return type;}

Partie<T>:
    std::list<unsigned> GetIds() const
    std::list<T> Get() const
```

I

```
ConstraintBuilder:
    inline bool IsBound() const

Expr:
    bool IsNum()
    bool IsVar()

Int:
    Int(int _entier = 0)

IntVar:
    IntVar( Model &_modele, int lowerBound = 0, int upperBound = Infinity,  →
       ↪ const std::string & nom = "unknow" )
    IntVar(Model & _modele, const std::string & _nom = "unknow" )

IntVarArray:
    IntVarArray( Model &_modele, unsigned size=0, int _lb= 0, int _ub= Infinity →
       ↪ ,  const std::string & nom = "unknow", const std::string & fin = "")

IntVarFamily<N,T>:
    IntVarFamily( Model &_modele, int _lb, int _ub, const Ensemble< Nuplet<N,T →
       ↪ > > & ensemble, const std::string & nom )

IntVarMatrix:
    IntVarMatrix( Model &_modele, unsigned sizeI=0, unsigned sizeJ=0, int _lb →
       ↪ = 0, int _ub= Infinity,  const std::string & nom = "unknow", const  →
       ↪ std::string & milieu = "", const std::string & fin = "" )

Model:
    void InitSolver(API _API=GLPK,const std::string& _fLP="./tmp.solver.lp",  →
       ↪ const std::string& _path="")
```

```
    bool IsRelaxed() const

NumVar:
    bool IsLowerBoundStrict() const
    bool IsUpperBoundStrict() const
```

L

```
LagrangianParam:
    LagrangianParam(Model & _model, std::string constraintName )
    LagrangianParam(Model & ,VarId)

Model:
    bool LoadLP(const std::string& , bool verbose = false)
```

M

```
Model:
    double MatriceOccupation()
    void Maximize(const Expr & expr, const std::string & nom = "")
    void Minimize(const Expr & expr, const std::string & nom = "")
```

N

```
Model:
    void Normalize(bool withoutStrict = true)

Nom:
    Nom(const std::string& _var)
    Nom(const std::string& _var, const char _i    )
    Nom(const std::string& _var, const double _i  )
    Nom(const std::string& _var, const float _i   )
    Nom(const std::string& _var, const int _i     )
    Nom(const std::string& _var, const long _i    )
    Nom(const std::string& _var, const unsigned _i)
    Nom(const std::string& _var, const char _i    , const char _j    )
    Nom(const std::string& _var, const double _i  , const double _j  )
    Nom(const std::string& _var, const float _i   , const float _j   )
    Nom(const std::string& _var, const int _i     , const int _j     )
    Nom(const std::string& _var, const long _i    , const long _j    )
    Nom(const std::string& _var, const unsigned _i, const unsigned _j)
    Nom(const std::string& _var, const char _i    , const std::string& _c, →
        ↪ const char _j    )
    Nom(const std::string& _var, const double _i  , const std::string& _c, →
        ↪ const double _j  )
    Nom(const std::string& _var, const float _i   , const std::string& _c, →
        ↪ const float _j   )
    Nom(const std::string& _var, const int _i     , const std::string& _c, →
        ↪ const int _j     )
```

```
    Nom(const std::string& _var, const long _i    , const std::string& _c,  →
        ↪ const long _j    )
    Nom(const std::string& _var, const unsigned _i, const std::string& _c,  →
        ↪ const unsigned _j)

Num:
    Num(int entier)
    Num(unsigned int entier)
    Num(float _value)
    Num(double _value)
    Num & operator= ( float _value )

NumArray:
    NumArray(MuteVar::VarType _type = MuteVar::FLOAT, unsigned _size = 0,float  →
        ↪ _value = 0.0f )
    NumArray(const std::vector< Num > & _vect, MuteVar::VarType _type = MuteVar →
        ↪ ::FLOAT)
    NumArray(const std::vector< double > & _vect)
    NumArray(const std::vector< float > & _vect)
    NumArray(const std::vector< int > & _vect)
    NumArray(const std::vector< unsigned > & _vect)

NumVar:
    NumVar(Model & _modele, float _lb= 0, float _ub= Infinity, MuteVar::VarType →
        ↪  _type = MuteVar::FLOAT, const std::string & _nom = "unknow" )
    NumVar(Model & _modele, MuteVar::VarType _type = MuteVar::FLOAT,const std:: →
        ↪ string & _nom = "unknow" )
    NumVar(const Model *,VarId)

NumVarArray:
    NumVarArray( Model & _modele, unsigned size = 0, float _lb= 0, float _ub=  →
        ↪ Infinity, MuteVar::VarType _type = MuteVar::FLOAT, const std::string →
        ↪  & nom = "unknow", const std::string & fin = "")

NumVarFamily<N,T>:
    NumVarFamily(Model & m, float lb, float ub, MuteVar::VarType type, const  →
        ↪ Ensemble< Nuplet_T > & ensemble, const string &format)

NumVarMatrix:
    NumVarMatrix( Model & _modele, unsigned sizeI = 0, unsigned sizeJ = 0,  →
        ↪ float _lb= 0, float _ub= Infinity, MuteVar::VarType _type = MuteVar →
        ↪ ::FLOAT, const std::string & nom = "unknow",  const std::string &  →
        ↪ milieu = "", const std::string & fin = "" )
    NumVarMatrix( const std::vector<NumVarArray *> & nva )
```

O

```
Contraint:
    bool operator==( const Constraint & c1, const Constraint & c2)

ConstraintBuilder:
    ConstraintBuilder & operator== ( const ConstraintBuilder & )
    ConstraintBuilder & operator!= ( const ConstraintBuilder & )
    ConstraintBuilder & operator<= ( const ConstraintBuilder & )
    ConstraintBuilder & operator>= ( const ConstraintBuilder & )
    ConstraintBuilder & operator< ( const ConstraintBuilder & )
    ConstraintBuilder & operator> ( const ConstraintBuilder & )
    ConstraintBuilder & operator== ( const Expr & )
    ConstraintBuilder & operator!= ( const Expr & )
    ConstraintBuilder & operator<= ( const Expr & )
```

```
    ConstraintBuilder & operator>= ( const Expr & )
    ConstraintBuilder & operator< ( const Expr & )
    ConstraintBuilder & operator> ( const Expr & );
    ConstraintBuilder operator== ( const Expr &, const Expr & )
    ConstraintBuilder operator!= ( const Expr &, const Expr & )
    ConstraintBuilder operator<= ( const Expr &, const Expr & )
    ConstraintBuilder operator>= ( const Expr &, const Expr & )
    ConstraintBuilder operator< ( const Expr &, const Expr & )
    ConstraintBuilder operator> ( const Expr &, const Expr & )
    std::ostream & operator<< (std::ostream & flux,const ConstraintBuilder & c)

Expr:
    Expr & operator+=( const Expr & )
    Expr & operator-=( const Expr & )
    Expr & operator*=(const Num &);
    Expr operator*( const Num & , const NumVar &)
    Expr operator*( const NumVar & , const Num &)
    Expr operator*( const NumVarArray & , const NumArray &)
    Expr operator*( const NumArray & , const NumVarArray &)
    Expr operator*( const Expr &, const Num & )
    Expr operator*( const Num & , const Expr &)
    Expr operator+( const Expr &, const Expr &)
    Expr operator-( const Expr &, const Expr &)
    std::ostream & operator<< (std::ostream & flux,const Expr & expr);

LagrangianParam:
    LagrangianParam & operator= (float _value)
    LagrangianParam & operator+= (float _value)
    LagrangianParam & operator-= (float _value)

Model:
    std::ostream & operator<<(std::ostream&,const Model&)
    std::ostream & Out( std::ostream & flux = std::cout, const std::string &  →
       ↪ newline = "" )
    std::ostream & OutConstraintValues(std::ostream & flux = std::cout, const  →
       ↪ std::string & newline = "" ) const
    std::ostream & OutDimensions( std::ostream & flux = std::cout, const std:: →
       ↪ string & newline = "" )
    std::ostream & OutNumVarValues( bool withoutNullValue = true,std::ostream →
       ↪  & flux = std::cout, const std::string & newline = "" ) const


Nom:
    std::string operator()() const
    Nom& operator<<(const char _i    )
    Nom& operator<<(const double _i  )
    Nom& operator<<(const float _i   )
    Nom& operator<<(const int _i     )
    Nom& operator<<(const long _i    )
    Nom& operator<<(const std::string& _str)
    Nom& operator<<(const unsigned _i)

NumArray:
    float operator[] (unsigned i) const
    Num & operator[] (unsigned i)
    NumArray & operator+=( const NumArray & na)
    NumArray & operator-=( const NumArray & na)
    NumArray & operator*=( Num f)

NumVar:
    inline bool operator==( const NumVar & n1, const NumVar & n2)

NumVarArray:
    NumVar& operator[] (unsigned n) const
```

```
NumVarMatrix:
    NumVar  operator() (unsigned i,unsigned j) const

Nuplet<N,T>:
    std::ostream & operator<< (std::ostream & flux,const Nuplet<N,T> & nuplet)
    std::istream & operator>> (std::istream & flux, Nuplet<N,T> & nuplet)
    T operator[] (unsigned n) const
    T & operator[] (unsigned n)


Omega:
    bool operator() (const Nuplet<N,T> & n) const

Partie<T>:
    Partie & operator= (const Ensemble<T> & _ens)
    Partie<T> operator& (const Partie<T>&a, const Partie<T> &b)
    Partie<T> operator| (const Partie<T> &a, const Partie<T> &b)
    Partie<T> operator/ (const Partie<T> &a, const Partie<T>& b)
    Partie<T> operator/ (const Ensemble<T>& e, const Partie<T> & b)
    std::ostream & operator<< (std::ostream & flux, const Ensemble<T> & ens)
    std::istream & operator>> (std::istream & flux, Ensemble<T> & ens)
    std::ostream & operator<< (std::ostream & flux,const Partie<T> & p)
```

P

```
Ensemble<T>
    Partie partie( const Propriete<T> & p)

Model:
    void PrintDual(const std::string&) const
    void PrintFraction()
    void PushOptions(const std::string& _str)

Partie<T>:
    Partie(Ensemble<T> * _ens = 0 )

Propriete<T>:
  virtual bool operator() (const T &) const= 0;
```

R

```
Constraint:
    bool Relax( float lagrangianValue = -Infinity)
    void Rename( const std::string & name )

Model:
    bool Relax(const std::string & name, float lagrangianValue = -Infinity)
    bool RelaxBool2Float( NumVar numVar = NumVar() )
    bool RelaxConstraint( VarId id, float lagrangianValue = -Infinity )
    bool RelaxInt2Float ( NumVar numVar = NumVar() )
    void RenameConstraint( VarId id, const std::string & name)
    void RemoveConstraint(VarId varId)
    void RemoveOptions()
    void RemoveVariable(VarId varId)
```

S

```
Constraint:
    void SetSecondMember(float)

Ensemble<T>
    unsigned Size() const

Expr:
    unsigned Size()
    Expr Sum(const NumVarArray & )

LagrangianParam:
    void SetValue( float )

Model:
    void SetAPI(API _API)
    void SetComment( VarId id, const std::string &  comment)
    void SetConstraintValues(const std::list<std::string> & noms, const std:: →
       ↪ list<float> & values)
    void SetNumVarValues(const std::list<std::string> & noms, const std::list< →
       ↪ float> & values)
    void SetProblemName( const std::string name = "")
    void SetRelaxed(bool _isRelaxed)
    void SetSecondMember(VarId,float)
    void ShowMe(  bool withoutNullValue = true )
    float Solve()
    void SolveInfos(std::string& _operation, std::string& _informations)
    void SortieHTML( const std::string & fileName, const std::string& matrixPNG →
       ↪ ="")
    void SortieLATEX( const std::string & fileName)
    void Standardize(bool withoutStrictConstraintConversion = true,bool  →
       ↪ withOnlyPositiveVariable = true)

Nom:
    void SetNom(const std::string& _nom)

Num:
    void SetType(MuteVar::VarType _type)  { type = _type; }
    void SetValue( float f) { *this = f;}

NumArray:
    unsigned Size() const { return cstArray.size();}

NumVar:
    void SetLowerBound(float _value)
    void SetLowerBoundStrict( bool _value = true )
    void SetName(const std::string &)
    void SetType( MuteVar::VarType _type)
    void SetUpperBound(float _value)
    void SetUpperBoundStrict( bool _value = true )

NumVarArray:
    unsigned Size() const

NumVarFamily<N,T>:
    unsigned Size() const
```

```
Partie<T>:
    unsigned Size() const
```

U

```
Model:
    void UnrelaxModel();
```