Programmer en C++ avec Cplex : *Concert Technology*

Les bases

Hélène Toussaint, mardi 2 juin 2015

Prérequis : une bonne connaissance du C++ et de la programmation linéaire.

Ce document présente de manière synthétique et non exhaustive les principales caractéristiques de la librairie *Concert Technology* pour le C++. Il est divisé en 2 parties :

- la **première partie** (sections 1 à 6) donne les bases nécessaires pour créer un PL (ou un PLNE) et le résoudre avec Cplex. Il s'agit du strict minimum à connaître pour utiliser les fonctions basiques de la *Concert Technology.*
- la **seconde partie** (sections 7 à 11) présente quelques spécificités de la *Concert Technology* comme les types et les classes propres à cette librairie. La plupart des notions présentées dans cette partie ne sont pas indispensables à un programme *Concert Technology* mais elles en facilitent grandement l'écriture et la gestion des erreurs.

Le but de ce document est de donner au lecteur une idée des différentes fonctionnalités de la *Concert Technology* mais sans entrer dans les détails. Chaque type, classe ou fonction présenté ici est détaillé dans le manuel de référence disponible en ligne ([1]). Il existe également une documentation pdf plus synthétique ([2]). Une documentation et des exemples sont également fournis dans le répertoire d'installation de **Cplex**.

TABLE DES MATIÈRES

PARTIE 1 : PRESENTATION GENERALE

Dans cette première partie on donne les bases indispensables à l'utilisation de la *Concert Technology* : création d'un modèle (i.e. PL ou PLNE), résolution du modèle, accès à la solution et compilation. Cette partie se termine par un exemple de programme complet.

1 *CONCERT TECHNOLOGY* : PRESENTATION

Concert Technology est un ensemble de librairies permettant d'intégrer les fonctionnalités de **Cplex** à un programme écrit en C++, Java ou .NET. Il s'agit d'une librairie composée de classes, son utilisation requiert donc une programmation orientée objet. Dans ce document on ne présente que la version pour C++. Les exemples sont écrits avec la version 12 de **Cplex**.

2 INCLUSION DE LA LIBRAIRIE ET COMPILATION

Les fichiers C++ doivent inclure la librairie **Cplex** et la macro `ILOSTLBEGIN` qui remplace la directive **`using namespace std`** :

```
#include <ilcplex/ilocplex.h>
ILOSTLBEGIN
```

A la compilation des flags supplémentaires sont nécessaires pour *linker* avec la librairie Cplex (voir http://www.isima.fr/~toussain/index.php?option=com_content&view=article&id=13&Itemid=15). Les exemples fournis à l'installation de **Cplex** contiennent des *makefile* (sous linux) ou des projets Visual Studio (sous Windows) déjà configurés correctement pour compiler avec Cplex.

Attention sous Windows : toutes les versions de **Cplex** ne sont pas compatibles avec toutes les versions de Visual Studio. Pour savoir avec quelles versions de Visual Studio votre version de **Cplex** est compatible il faut regarder dans le répertoire <REP_D_INSTALL>\concert\lib, vous y trouverez des répertoires contenant les .lib à inclure dans vos projets Visual. Ces répertoires portent le nom de la version de Visual avec laquelle les .lib sont compatibles.

3 LES TROIS PRINCIPALES CLASSES DE LA CONCERT TECHNOLOGY

Concert Technology est constituée d'un ensemble de classes. Les noms de types et de classes sont tous préfixés par **`Ilo.`**

Les classes indispensables à l'écriture d'un programme utilisant la *Concert Technology* :

- **`IloEnv`** : classe qui permet d'intégrer l'environnement **Cplex** à notre programme. Déclarer un objet de type `IloEnv` est la première chose à faire lors de l'écriture d'un programme qui utilise la *Concert Technology*. A la fin du programme il faut libérer l'environnement en appelant la méthode `end()` sur l'objet `IloEnv`.

Extrait de [3] : "*L'objet environnement est d'importance centrale et doit être accessible au constructeur de toutes les autres classes Concert Technology car (entre autres choses) il assure une gestion optimisée de la mémoire pour les objets des classes Concert Technology. Comme la plupart des classes Concert Technology, IloEnv est une classe de descripteur. Cela signifie que la variable* `env` *est un pointeur vers un objet d'implémentation.*"

- **`IloModel`** : classe qui regroupe les différents objets (contraintes, variables, fonction objectif, ...) associés à un problème d'optimisation. Une instance d'`IloModel` peut être vue comme la représentation du problème d'optimisation.

- **`IloCplex`** : classe qui regroupe les fonctions du solveur **Cplex** (paramétrer le solveur, résoudre le problème, accéder à la solution, ...). Une instance d'`IloCplex` permet de lire une instance d'`IloModel` et en extrait les informations nécessaires à la résolution du modèle par le solveur.

```
IloEnv          env; //variable d'environnement
IloModel        mod ( env ); //modèle, on l'associe directement à l'environnement
IloCplex        cplex ( mod ); //solveur, on l'associe directement au modèle

//coeur du programme : construction du modèle, résolution, etc ...
//......

env.end(); //libération de l'environnement
```

Les déclarations des objets indispensables à une application *Concert Technology*

4 CONSTRUCTION D'UN MODÈLE COMPLET

Après avoir créé un objet de type `IloModel` (initialement vide) il faut lui intégrer les variables, contraintes et la fonction objectif qui formeront le modèle à optimiser.

La *Concert Technology* offre différentes possibilités pour faire cela. On voit dans cette partie comment créer les objets (i.e. les variables, contraintes, ...) un par un. Les différents objets doivent être ajoutés au modèle à l'aide de la méthode `add()`. Les classes servant à instancier ces objets disposent généralement de plusieurs constructeurs (avec souvent des paramètres par défaut) on n'en présente qu'un par classe (pour la liste complète voir [1]).

4.1 LES VARIABLES

La classe modélisant les variables est `IloNumVar`. Elle crée et stocke différentes informations sur la variable qu'elle représente (type, bornes, ...). Pour créer un objet de type `IloNumVar` on peut utiliser le constructeur suivant :

```
IloNumVar(const IloEnv     env, //pointeur sur l 'environnement
          IloNum           lb = 0, //borne inf. (par défaut 0)
          IloNum           ub = IloInfinity, //borne sup. (par défaut + l'infini)
          IloNumVar::Type type = Float, //type de la variable (par défaut réelle)
```

```
        const char *    name = 0 //nom de la variable (par défaut aucun)
        )
```

- Le premier paramètre est le pointeur sur l'environnement (comme pour presque toutes les fonctions de la *Concert Technology*)
- le second et le troisième sont du type `IloNum` (c'est le type qui représente les "`double`" dans la *Concert Technology*), ils fixent la borne inférieure et supérieure de la variable. La constante `IloInfinity` permet de fixer une borne infinie (i.e. variable non bornée)
- le quatrième fixe le type de la variable. Les types peuvent être définis de deux manières :
  - en utilisant les champs de la structure énumérative : `IloNumVar::Int`, `IloNumVar::Float`, `IloNumVar::Bool`
  - en utilisant les macros associées : `ILOINT`, `ILOFLOAT`, `ILOBOOL`.
- le dernier est le nom de la variable.

```
IloNumVar x1 (env, 0.0, 40.0, ILOFLOAT, "x1");
IloNumVar x2 (env, -10, 20, ILOINT, "x2");
```

Exemple : déclaration de deux variables (`env` est du type `IloEnv`, préalablement déclaré)

4.2 LES CONTRAINTES

La classe `IloRange` permet de définir des contraintes de la forme :

borne inférieure ≤ expression ≤ borne supérieure

Il existe de nombreuses manières de définir les contraintes d'un problème, on voit ici comment faire en utilisant un des constructeurs d'`IloRange` :

```
IloRange(const IloEnv        env, //pointeur sur l 'environnement
         IloNum              lhs, //borne. inf
         const IloNumExprArg expr, //expression
         IloNum              rhs = IloInfinity, //borne. sup
         const char *        name = 0 //nom de la contrainte
         )
```

```
//1ere contrainte : 2.5 x1 + x2 >= 2
IloRange ctr1 ( env, 2, 2.5 * x1 + x2, IloInfinity, "C1");

//2sd contrainte : x1 - 3 x2 <= 10
IloRange ctr2 ( env, -IloInfinity, x1 - 3 * x2, 10, "C2");
```

4.3 LA FONCTION OBJECTIF

La classe permettant de représenter la fonction objectif est `IloObjective`. Une fonction objectif est définie par un sens d'optimisation (minimisation ou maximisation) et une quantité à optimiser : somme pondérée de variables ou une constante. Afin de créer une instance de cette classe on peut utiliser indifféremment un constructeur de la classe ou une fonction qui se charge de construire l'objet et le retourne.

- **Utilisation du constructeur** : Le constructeur suivant crée une fonction objectif à partir d'une expression arithmétique passée en paramètre (NB : **IloCplex** définit un type spécial pour les expressions arithmétiques : `IloNumExprArg`, on reviendra plus en détail dessus dans la partie 2). Ce constructeur permet de préciser le sens d'optimisation (`IloObjective::Minimize` ou `IloObjective::Maximize`) et un nom éventuel pour la fonction objectif.

```
IloObjective( const IloEnv         env, //pointeur sur l 'environnement
              const IloNumExprArg expr, //expression arithmétique
              IloObjective::Sense sense = IloObjective::Minimize, //sens
              const char *        name = 0 //nom de la fonction objectif
            )
```

Un constructeur de la classe `IloObjective`

```
IloObjective obj (env, x1 + x2, IloObjective::Minimize, "OBJ");
```

Exemple : création d'une fonction objectif en utilisant un constructeur d'`IloObjective`

- **Utilisation de fonctions retournant un objet de type `IloObjective`** : deux fonctions `IloMinimize` et `IloMaximize` permettent de créer facilement une instance de `IloObjective` avec un sens d'optimisation indiqué par le nom de la fonction :

```
public IloObjective IloMinimize (const IloEnv env, //pointeur sur l 'environnement
                                 const IloNumExprArg expr, //expression arithmétique
                                 const char * name=0 //nom de la fonction objectif
                                )

public IloObjective IloMinimize (const IloEnv env, //pointeur sur l 'environnement
                                IloNum constant=0.0, //constante
                                const char * name=0 //nom de la fonction objectif
                                )
```

```
IloObjective obj = IloMinimize ( env, x1 + x2, "obj" );
```

Exemple : création d'une fonction objectif en utilisant la fonction `IloMinimize`

5 RESOLUTION DU MODELE ET AFFICHAGE DES RESULTATS : ILOCPLEX

L'optimisation, l'export du modèle dans un fichier, l'accès à la solution,... sont pris en charge par la classe `IloCplex` : après avoir construit et stocké le problème d'optimisation dans un objet `IloModel`, il faut créer un objet `IloCplex` et l'associer au modèle à résoudre (plusieurs modèles peuvent coexister dans une application). Pour cela il existe plusieurs possibilités :

- soit on déclare une instance d'`IloCplex` en passant directement au constructeur l'instance d'`IloModel` à laquelle on souhaite l'associer (voir section 3).
- soit on construit d'abord un objet `IloCplex` puis on l'associe ensuite à l'aide de la méthode `extract()` :

```
IloCplex cplex (env); // crée un objet IloCplex associé à aucun modèle
cplex.extract  (model); //associe l'objet IloCplex au modèle model
```

La classe `IloCplex` fournit un grand nombre de méthodes qui permettent d'accéder aux diverses fonctionnalités de Cplex. La plupart de ces méthodes acceptent des arguments par défaut et sont surchargées (voir [4]). On ne présente ici qu'une version par méthode.

- **`public IloBool solve()`** : résout le modèle courant et renvoie `IloTrue` si une solution (pas forcément optimale) a été trouvée (`IloFalse` sinon).
- **`public IloAlgorithm::Status getStatus()`** `const` : retourne le statut (de type `IloAlgorithm::Status`) de la solution courante. `IloAlgorithm::Status` est une énumération contenant les champs :
  - `Unknown` : pas d'information sur la solution
  - `Feasible` : la solution est réalisable (respecte toutes les contraintes) mais pas nécessairement optimale.
  - `Optimal` : la solution est optimale
  - `Infeasible` : il n'existe pas de solution au modèle courant
  - `Unbounded` : le modèle courant est non borné
  - `InfeasibleOrUnbounded` : le modèle courant est soit irréalisable soit non borné
  - `Error` : une erreur s'est produite lors de l'optimisation
- **`public void exportModel(const char * filename) const`** : sauvegarde le modèle courant dans le fichier spécifié en paramètre. Le format de fichier dépend de l'extension du nom de fichier fourni en paramètre (`.lp`, ou `.mps` par exemple)
- **`public IloNum getValue(const IloNumVar var) const`** : retourne la valeur de la variable `var` dans la solution courante
- **`public IloNum getObjValue( ) const`** : retourne la valeur de la fonction objectif de la solution courante.

6 EXEMPLE DE PROGRAMME COMPLET

```
#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

int main(int argc, char **argv)
{
    //déclaration des trois classes principales
    IloEnv env;             // environnement : permet d utiliser les fonctions de Concert Technology
    IloModel model(env);    // modèle : représente le programme linéaire
    IloCplex cplex (model); // classe Cplex : permet d 'accéder aux fonctions d'optimisation

    // ----------------------  VARIABLES  ----------------------
    //déclaration de 2 variables
    IloNumVar x1 (env, 0.0, 40.0, ILOFLOAT, "X1");
    IloNumVar x2 (env, -10, 20, ILOINT, "X2");
    //ajout des variables au modèle
    model.add(x1);
    model.add(x2);

    // ----------------------  OBJECTIF  ----------------------
    //déclaration de la fct objectif
```

```
    IloObjective obj (env, x1 + x2, IloObjective::Minimize, "OBJ");
    //ajout de la fct objectif au modèle
    model.add(obj);

    // ----------------------  CONTRAINTES  ----------------------
    //déclaration de deux contraintes
    IloRange ctr1 ( env, 2, 2.5 * x1 + x2, IloInfinity, "C1");
    IloRange ctr2 ( env, -IloInfinity, x1 - 3 * x2, 10, "C2");
    //ajout des contraintes au modèle
    model.add(ctr1);
    model.add(ctr2);

    // ------------------ AFFICHAGE ET OPTIMISATION ----------------
    // export du PL créé dans un fichier .lp
    cplex.exportModel("test.lp");
    // résolution
    cplex.solve();
    // export de la solution dans un fichier texte
    cplex.writeSolution("sol.txt");
    // récupère la solution et l'affiche à l'écran
    cout << " objectif = " << cplex.getObjValue() << endl;
    cout << " X1 = " << cplex.getValue(x1) << endl;
    cout << " X2 = " << cplex.getValue(x2) << endl;

    env.end();

    return 0;
}
```

Programme complet utilisant la *Concert Technology*, pour des raisons de lisibilité la gestion des erreurs n'est pas montrée

```
\ENCODING=ISO-8859-1
\Problem name: IloCplex

Minimize
 OBJ: X1 + X2
Subject To
 C1: 2.5 X1 + X2 >= 2
 C2: X1 - 3 X2 <= 10
Bounds
 0 <= X1 <= 40
-10 <= X2 <= 20
Generals
 X2
End
```

Fichier .lp associé au programme précédent

PARTIE 2 : FONCTIONS AVANCEES

La première partie permet d'écrire un programme de manière "naïve". On voit dans cette seconde partie des fonctionnalités plus avancées ainsi que différentes possibilités offertes par la *Concert Technology* pour faciliter la programmation.

7 GESTION DES ERREURS

7.1 LES ERREURS DE PROGRAMMATION

Les erreurs de programmation (objet non initialisé, mauvaise valeur pour l'argument d'une fonction, ...) sont traités par *Concert Technology* avec des instructions `assert`, de sorte que les vérifications associées ne sont effectuées qu'en mode **`DEBUG`**. En mode release (option de compilation **`-DNDEBUG`** activée) ces vérifications ne sont pas activées afin de ne pas ralentir l'exécution.

7.2 LES ERREURS SURVENANT PENDANT L'OPTIMISATION

Concert Technology émet une exception si une erreur survient pendant l'optimisation (manque de mémoire par exemple). Il existe une hiérarchie de classes d'exception toutes dérivées de la classe de base commune `IloException`.

Afin de gérer correctement les exceptions dans une application *Concert Technology*, il faut inclure une directive `try / catch` :

```
IloEnv env; //environnement, à definir avant le try / catch
try
{
    // programme
}
catch (IloException & e) //gestion des exceptions levées par Concert Technology
{
    cerr << " Exception de la Concert Technology " << e << endl;
}
catch (...) // les autres exceptions éventuelles levées dans le programme
{
    cerr << " Exception inconnue "  << endl;
}
env.end(); //toujours libérer l'environnement, même si on lève une exception
```

8 LES TYPES ET VARIABLES DANS CONCERT TECHNOLOGY

La *Concert Technology* définit ses propres constantes et types de variables (dans un souci de portabilité). Néanmoins on est libre d'utiliser les constantes et types de bases du C++.

Les constantes :

- `IloInfinity` : Constante représentant le plus grand nombre flottant de la plateforme. Si elle est utilisée comme borne pour une variable ou contrainte alors cela signifie que cette dernière est non bornée. Elle est initialisée lors de la construction d'un `IloEnv.`
- `IloIntMax (IloIntMin)` : constante définissant la plus grande (resp. plus petite) valeur que peut prendre un entier

Les redéfinitions des types de base :

- `IloInt` : redéfinit le type entier long
- `IloBool` : redéfinit le type booléen (peut prendre les valeurs `IloFalse` (0), et `IloTrue` (1))
- `IloNum` : redéfinit le type double

9 LES TABLEAUX DANS CONCERT TECHNOLOGY

9.1 LES TABLEAUX *TEMPLATE* : LA CLASSE ILOARRAY

Concert Technology possède une classe *template* pour créer des tableaux d'objets : `IloArray`. Elle s'utilise dans la même logique que la classe `vector` de la **STL**. On peut l'utiliser pour créer des tableaux à plusieurs dimensions en créant des `IloArray` d'`IloArray`.

Un *IloArray* est toujours associé à un environnement *IloEnv*. Le constructeur permet d'allouer un *IloArray* d'une taille donnée (0 par défaut) :

```
public IloArray(IloEnv env, IloInt max=0)
```

Si la taille n'est pas connue au moment de la déclaration de l'*IloArray*, on peut déclarer un tableau vide (de taille 0) et le remplir par la suite à l'aide de la méthode :

```
public void add(X x) const
```

qui ajoute un élément à la fin de l'*IloArray* (et le redimensionne donc en conséquence - comme la méthode `push_back()` du `vector` de la **STL**).

L'opérateur << est surdéfini pour la classe `IloArray`. On peut donc afficher en une ligne de code le contenu complet d'un tableau `IloArray` (et de toute classe dérivant d'un `IloArray`).

9.2 LES TABLEAUX D'OBJETS DEJA DEFINIS

Concert Technology définit des classes "tableaux" pour chaque classe de base `X` qu'elle possède (elles utilisent la classe `IloArray` vue précédemment). La classe s'appelle alors `IloArrayX` ou `IloXArray.` Par exemple pour créer un tableau d'`IloNumVar` il existe la classe `IloNumVarArray`. Elle possède plusieurs constructeurs qui permettent de créer en une ligne un ensemble de variables (avec les mêmes caractéristiques ou avec des caractéristiques différentes). On dispose par exemple d'un constructeur qui crée un tableau de *n* variables ayant toutes les mêmes bornes et le même type (`ILOFLOAT` par défaut) :

```
public IloNumVarArray(const IloEnv env, //environnement
                          IloInt n, //nombre de variables a créer (= taille du tableau)
                          IloNum lb, // borne inf (commune a toutes les variables)
                          IloNum ub, // borne sup (commune a toutes les variables)
                          IloNumVar::Type type=ILOFLOAT //type (commun a toutes les var.)
                      )
```

Si on souhaite imposer des bornes différentes aux variables il existe un second constructeur qui prend en paramètres un premier tableau contenant les bornes inférieures et un second tableau contenant les bornes supérieures. Il est clair que ces deux tableaux doivent avoir la même taille, taille qui sera également celle du tableau de variables créé :

```
public IloNumVarArray(const IloEnv env, const IloNumArray lb, const IloNumArray ub,
IloNumVar::Type type=ILOFLOAT)
```

Par ailleurs on peut aussi, dans un premier temps, déclarer des tableaux vides et, dans un second temps, affecter une variable à chaque case du tableau. L'exemple suivant illustre cette possibilité en créant un tableau de tableaux de variables pour représenter des variables à double indices ($n \times m$ variables booléennes x_{ij} $(i = 0, \dots, n-1\,; j = 0, \dots, m-1)$) :

```
   IloArray<IloNumVarArray> x_ij (env, n);
   for ( int i = 0; i < n; i++ )
   {
        //chaque x_ij[i] est un tableau de m variables booléennes
        x_ij[i] = IloNumVarArray(env, m, 0.0, 1.0, ILOBOOL);
   }
```

10 LES EXPRESSIONS NUMERIQUES DANS CONCERT TECHNOLOGY

Concert Technology dispose d'une classe très pratique pour représenter et manipuler les expressions numériques (combinaison linéaire de variables par exemple) : `IloExpr`. On écrit dans une `IloExpr` les expressions de manière naturelle :

```
IloNumVar x (env); // crée une variable avec les paramètres par défaut (float, >= 0)
IloNumVar y (env);
IloExpr expr = x + 2*y;
```

On peut facilement ajouter ou retirer des termes dans une `IloExpr` déjà initialisée en utilisant les opéteur `+=` et `-=`. Ces derniers sont très pratiques dans le cas où l'expression est construite à l'aide d'une itérative.

Il y a deux règles à respecter :

- Une `IloExpr` doit toujours être initialisée (avant d'utiliser un opérateur tel que `+=` par exemple)
- Une `IloExpr` doit toujours être libérée en appelant la méthode `end()` lorsqu'on n'en a plus besoin

`IloExpr` dérive de `IloNumExpr` qui dérive elle-même de `IloNumExprArg`. On peut donc l'utiliser comme argument dans les méthodes qui attendent un argument de type `IloNumExprArg` comme c'est le cas par exemple pour les constructeurs d'`IloRange` ou d'`IloObjective` vus précédemment.

11 PARAMETRAGE DES ALGORITHMES

La classe IloCplex dispose d'une méthode :

```
public void setParam(<nom du paramètre>, <valeur du paramètre>)
```

qui permet de fixer la valeur des paramètres de Cplex. Il existe un grand nombre de paramètres de types différents (booléen, entier, réel, chaîne de caractères, ...), la méthode `setParam` est surdéfinie pour chacun. Dans le cas où aucune valeur n'a été attribuée à un paramètre via `setParam()` la valeur par défaut du paramètre est utilisée par **Cplex**.

Les paramètres sont définis dans des énumérations imbriquées dans la classe `IloCplex`, il faut donc toujours les préfixer par `IloCplex::`.

Quelques exemples d'utilisation de `setParam()` :

```
//cpu max (en secondes)
cplex.setParam(IloCplex::TiLim, 10);

//desactive l'affichage lors de la résolution d'un MIP
cplex.setParam(IloCplex::MIPDisplay, 0);

//limite le nombre de threads utilise par cplex (32 par défaut)
cplex.setParam(IloCplex::Threads, 2);
```

Cplex dispose de différents algorithmes et on peut choisir lequel on souhaite utiliser. Par défaut **Cplex** résout le modèle courant comme un MIP, et nous autorise donc à choisir un algorithme pour la racine et un autre pour les nœuds :

```
void IloCplex::setParam(IloCplex::RootAlg, algo)
void IloCplex::setParam(IloCplex::NodeAlg, algo)
```

où `algo` est à choisir parmi l'énumération `IloCplex::Algorithm` :

- **`IloCplex::AutoAlg`** : algorithme choisi par Cplex
- **`IloCplex::Primal`** : simplexe primal
- **`IloCplex::Dual`** : simplexe dual
- **`IloCplex::Network`** : network simplex algorithm
- **`IloCplex::Barrier`** : *barrier algorithm* (souvent efficace sur les grandes matrices creuses)
- **`IloCplex::Sifting`** : *sifting algorithm*
- **`IloCplex::Concurrent`** : permet a **Cplex** d'utiliser différents algorithmes sur les machines multiprocesseurs

Pour connaître l'algorithme utilisé par Cplex pour résoudre le modèle courant (si ce n'est pas un MIP) on peut utiliser la fonction :

```
public IloCplex::Algorithm getAlgorithm() const
```

12 CHRONOMÈTRE

Concert Technology dispose d'une classe `IloTimer` qui fonctionne comme un chronomètre. Pour l'utiliser on a besoin des trois méthodes suivantes :

```
//constructeur
public IloTimer(const IloEnv env)

//initialise le chronometre
public IloNum start ()

//retourne le temps écoulé depuis l'appel à start() en secondes (si on est sur un
environnement multithreadé, le temps des différents threads est cumulé)
public IloNum getTime () const
```

13 QUELQUES FONCTIONS MATHEMATIQUES DE CONCERT TECHNOLOGY

Concert Technology dispose de fonctions qui peuvent être très pratiques lors de la construction d'un modèle. On en donne ici un aperçu :

- **produit scalaire** entre deux *arrays* de même taille :

```
public IloNumExprArg IloScalProd ( const IloNumArray values,
                                   const IloNumVarArray vars)
```

  On remarque que cette fonction retourne une `IloNumExprArg`, on peut donc l'ajouter directement au modèle. Par exemple si on a initialisé un tableau de coefficients `coeffs` et un tableau de variables `vars`, on peut écrire :

```
model.add( IloScalProd(coeffs, vars) <= 20 )
```

- **arrondis** :
  - à l'entier supérieur : `public IloNum IloCeil(IloNum val)`
  - à l'entier inférieur : `public IloNum IloFloor(IloNum val)`
  - à l'entier le plus proche : `public IloNum IloRound(IloNum val)`

- **sommes :**
  - somme de nombres : `public IloNum IloSum(const IloNumArray values)`
  - somme de variables : `public IloNumExprArg IloSum(const IloNumVarArray exprs)`

14 UN PROGRAMME MINIMAL

On a vu dans la première partie comment déclarer variables et contraintes. En réalité on n'est pas obligé de les déclarer explicitement (c'est surtout utile si on a besoin de les stocker pour s'en servir par la suite). On peut simplement les ajouter au modèle sous forme d'*Ilo expressions* comme le montre l'exemple suivant :

```
#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

int main()
{
   IloEnv   env;
   try
   {
      IloModel model(env, "exemple2");

      //déclaration d'un tableau de 3 variables (réelles) de borne inf. 0 et sup. 50
      IloNumVarArray x(env, 3, 0.0, 50.0);

      // ajout au modèle de la fonction objectif (somme des variables)
      model.add(IloMinimize(env, x[0] + x[1] + x[2]) );

      //ajout au modèle des contraintes
      model.add( -x[0] + x[1] + 3 * x[2] == 15);
      model.add( x[0] + 5 * x[1]  <= 20);
      model.add(  6 * x[1] - x[2] >= 15);

      //associe le modèle a une instance d'IloCplex pour résolution
      IloCplex cplex(model);
      //resolution
      cplex.solve();

      //résultats
      cout << " obj = " << cplex.getObjValue() << endl;
      IloNumArray vals(env);
      cplex.getValues(vals, x);
      cout << " variables =  " << vals << endl;

      cplex.exportModel("test.lp");

   }
   catch (IloException& e)
   {
      cerr << " ERREUR : exception = " << e << endl;
   }

   env.end();
   return 0;
}
```

Programme généré par le code précédent :

```
\ENCODING=ISO-8859-1
\Problem name: IloCplex

Minimize
 obj: x1 + x2 + x3
Subject To
 c1: - x1 + x2 + 3 x3  = 15
 c2: x1 + 5 x2 <= 20
 c3: 6 x2 - x3 >= 15
Bounds
```

```
 0 <= x1 <= 50
 0 <= x2 <= 50
 0 <= x3 <= 50
End
```

15 RÉFÉRENCES

[1] IBM Knowledge Center, Manuel de référence en ligne
http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.6.0/ilog.odms.cplex.help/refcppcplex/html/overview.html?cp=SSSA5P_12.6.0%2F2-8&lang=fr

[2] IBM, IBM ILOG CPLEX Optimization Studio Getting Started with CPLEX, Version 12 Release 4
http://cedric.cnam.fr/~lamberta/MPRO/ECMA/doc/Interface.pdf

[3] IBM Knowledge Center, Anatomie d'une application C++ Concert Technology
http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.6.0/ilog.odms.cplex.help/CPLEX/GettingStarted/topics/tutorials/Cplusplus/appliAnatomy_synopsis.html?lang=fr

[4] IBM Knowledge Center, La classe IloCplex
http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.6.0/ilog.odms.cplex.help/refcppcplex/html/classes/IloCplex.html?lang=fr