API CPLEX

CPLEX est un outil commercial d'optimisation mathématique. Il doit son nom à l'algorithme du simplexe, un algorithme relativement ancien pour les problèmes d'optimisation linéaire, mais il supporte actuellement bien d'autres types de programmes mathématiques (programmes entiers, mais également non linéaires – quadratiques). Il fait partie [des outils les plus performants à ce niveau](http://scip.zib.de/), de par sa rapidité de traitement.

Il dispose d'une interface interactive en ligne de commande (CPLEX Interactive Optimizer), mais également d'interfaces dans divers langages de programmation, dont C++ et Java (Concert Technology). Ce document se focalise sur ces interfaces.

[Une version d'évaluation de trois mois](http://www-01.ibm.com/software/websphere/products/optimization/cplex-studio-preview-edition/), grandement limitée dans la taille des problèmes, est disponible gratuitement ; pour le monde académique, [une licence gratuite sans limite](http://www-304.ibm.com/ibm/university/academic/pub/page/academic_initiative) est également proposée.

# Architecture d'une application

Une application utilisant CPLEX travaillera principalement avec un objet IloCplex ou IloEnv : il permet de créer des variables (IloNumVar) et des contraintes (IloConstraint), mais aussi d'appeler le solveur et de récupérer les solutions. Tous les objets créés ne seront que des interfaces vers la représentation interne du solveur CPLEX de ces variables et contraintes : le fonctionnement interne de CPLEX n'est pas montré, seule une version abstraite est disponible – et largement suffisante pour la plupart des besoins.

Une application se déroulera donc en trois phases :

* la création du modèle ;
* sa résolution ;
* l'utilisation de la solution.

Pour de petits modèles, il est envisageable de n'utiliser qu'une fonction qui gère l'entièreté du modèle, mais cette solution n'est pas applicable à grande échelle. Pour un modèle de taille plus conséquente, le paradigme objet, utilisé pour la conception de l'API Concert, se révélera souvent utile. Une analyse du problème à modéliser donne une liste de concepts qui y interviennent (par exemple, les ressources disponibles, les machines à utiliser, les objets à fabriquer) : chacun deviendra une classe, dont les instances seront responsables de la création des variables et des contraintes qui leur correspondent (une machine ne gérera que sa propre production).

# Création d'un modèle

La première étape est d'instancier CPLEX. En C++, il s'agira de créer un objet IloEnv, puis un modèle IloModel ; en Java, une instance de IloCplex gère l'entièreté du processus.

**Program listing. Language: c++.**

IloEnv env;  
IloModel model(env);

**Program listing. Language: java.**

IloCplex cplex = new IloCplex();

De là, il est possible de créer une variable, par exemple entière, qui prend ses valeurs entre zéro et quarante-deux, avec un nom "x" (il sera utile pour le débogage).

**Program listing. Language: c++.**

IloNumVar x(env, 0, 42, IloNumVar::Int, "x");

**Program listing. Language: java.**

IloNumVar x = cplex.intVar(0, 42, "x");

Il est également possible de créer des tableaux de variables, par exemple booléennes, dont les noms sont donnés par un tableau de chaînes de caractères (en Java).

**Program listing. Language: c++.**

IloNumVarArray y(env, 2, 0, 1, IloNumVar::Bool);

**Program listing. Language: java.**

String[] namesY = new String[] { "y1", "y2" };  
IloNumVar[] y = cplex.boolVarArray(2, namesY);

De là, l'API permet d'exprimer des contraintes entre ces variables, chacune avec un nom donné.

**Program listing. Language: c++.**

model.add(y[0] <= y[1], "c1");  
model.add(x <= 1000 \* y[1], "c2");

**Program listing. Language: java.**

cplex.addLe(y[0], y[1], "c1");  
cplex.addLe(y[0], cplex.prod(1000, y[1]), "c2");

La dernière étape est de créer un objectif.

**Program listing. Language: c++.**

model.add(IloMinimize(env, x));

**Program listing. Language: java.**

cplex.addMinimize(x);

# Optimisation d'un modèle

La résolution du modèle se réduit ensuite à l'appel d'une fonction.

**Program listing. Language: c++.**

IloCplex cplex(model);  
cplex.solve();

**Program listing. Language: java.**

cplex.solve();

La méthode solve() retourne un booléen vrai si l'optimisation s'est déroulée sans problème (si une solution a été trouvée). getStatus() donne plus de détail (solution optimale trouvée, solution faisable mais non optimale trouvée, modèle infaisable, etc.).

La valeur de l'objectif est accessible par la méthode getObjValue().

**Program listing.**

double objective = cplex.getObjValue();

La valeur d'une variable particulière s'obtient en passant son objet IloNumVar en paramètre à getValue().

**Program listing.**

double xV = cplex.getValue(x);

Pour récupérer les valeurs de plusieurs variables, getValues() prend en paramètre un tableau.

**Program listing. Language: c++.**

double yV[] = cplex.getValues(y);

**Program listing. Language: java.**

double[] yV = cplex.getValues(y);

# Résolution d'infaisabilité

En cas de problème dans la résolution (pas de solution faisable, loin de l'optimalité, etc.), il est bien souvent utile d'exporter le modèle créé. CPLEX permet de créer très facilement des fichiers LP contenant toutes les contraintes et objectifs, contenant notamment leur nom.

**Program listing.**

cplex.exportModel("model.lp");

Ce fichier peut alors être chargé dans l'interface interactive, où [un outil de détection de conflits](http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r6/index.jsp?topic=%2Filog.odms.ide.help%2Frefcppopl%2Fhtml%2Fconflict.html) peut se révéler précieux (il est également accessible par les API CPLEX).

Dans la console interactive, une fois le modèle chargé et optimisé, la commande conflict