CPLEX

Introduction

Check this first:

ILOG CONCERT	*Solvers*	
	CPLEX	mathematical programming (LP, QP, MILP, MIQP)
	CP	constraint programming

- A common way to define the model (CONCERT environment)
- All classes are tied to the environment
- Components (variables/constraints/objective) are added to the model
- Model is extracetd by CPLEX

Base classes to solve models:

	Steps	Classes
CONCERT	environment (main class)	**IloEnv**
	model	**IloModel**
	variables	**IloNumVar, IloNumColumn, IloNumVarArray, IloConversion**
	constraints	**IloRange, IloExpr, IloRangeArray**
	objective	**IloObjective**
CPLEX	solve	**IloCplex**

- All classes (except IloCplex) are listed in the Concert section of the reference guide (see link of online documentation at the bottom of this page).

- IloCplex class also has an "advanced" section with methods that give users an opportunity to interfere with the normal progress of branch&cut (=> callbacks)

IMPORTANT:

- By default, CPLEX (as Gurobi, Matlab and others do) will use **ALL** CPUs and cores of the machine on which the program runs.

- **To limit the number if threads used by CPLEX, we ask you to use this instruction to fix the limit to 1:**

```
cplex.setParam(IloCplex::Param::Threads, 1);
```

Interfaces:

Interactive	load a model from disk and solve it while fixing parameters manually
Library	build and solve a model with classes and functions in various languages (C, C++, Java, Python, .NET, MATLAB)
OPL	special language from ILOG to interact with Excel among other things

Compilation:

	Headers	Preprocessor directives (-D)
C	#include <ilcplex/cplex.h>	
C++	#include <ilcplex/ilocplex.h>	IL_STD ==> STL access NDEBUG ==> assert (checking disabled)
Java	import ilog.concert.*; import ilog.cplex.*;	

	Libraries
C	libcplex.a libpthread.a libm.a
C++	**libilocplex.a libconcert.a libcplex.a** libpthread.a libm.a
Java	cplex.jar

N.B.: In C++, the order of libraries listed in bold is IMPORTANT!!!

Macro (C++): **ILOSTLBEGIN** **(===> *using namespace std;)***

- Must be used before any call to CPLEX classes.
- In the future, could be extended to include ther features.

CPLEX C++ (Concert):

Overview :

- Classes, definitions and base types:

IloExtractable	base class for model building classes
IloAlgorithm	base class for model solving classes
IloNum, **IloInt** **IloBool**	alias for double, int, bool (coded as int)
ILOFLOAT **ILOINT** **ILOBOOL**	types for numeric, integer, binary variables
IloInfinity	infinity (bounds of variables/constraints)

- Environment and model

IloEnv	create a common environment (build + solve model)
IloModel	create LP, QP, MILP, MIQP model (with variables, constraints, etc.)

- Objective:

IloObjective	define objective function
IloMinimize, IloMaximize	sets optimization direction (*min* / *max*)

- Variables:

IloNumColumn	populate variables adding coefficients one by one
IloNumVar **IloIntVar** **IloBoolVar**	declare numeric (defaut), integer, binary variables
IloConversion	convert numeric variables in other types

- Constraints:

IloExpr	build an expression adding coefficients one by one
IloRange	declare constraints
IloSum	return the sum of variables
IloScalProd	return the scalar product between an array of variables and an array of coefficients
IloIfThen	create "Big M" or logic constraints

- Arrays + macro:

IloArray	Class template to create multi-dimension extensible arrays
IloExtractableArray	Extensible array of IloExtractable
IloNumArray **IloIntArray** **IloBoolArray**	Extensible arrays of base types (IloNum, IloInt, IloBool)
IloNumVarArray **IloIntVarArray** **IloBoolVarArray**	Extensibles arrays of variables (IloNumVar, IloIntVar, IloBoolVar)
IloRangeArray	Extensibles arrays of constraints
IloAdd	Macro to define objects and add them to the model in one shot.

- Solve:

IloCplex	Solve model with CPLEX

Base classes:

- **IloExtractable** : base class for model building classes

end()	destroy the element (replace the destrcutor)
getEnv()	retrieve the environment to which the element is tied
removeFromAll()	remove element from all other related objects
setName(const char*)	give name to the element
getObject()	retrieve "object" assigned to the element
setObject(IloAny)	assign object to element

- **IloAlgorithm** : base class for model solving classes

clear()	reinitialize solver
end()	destroy solver (replace destructor)
extract(const IloModel)	load model in the solver
setError(ostream&) **setWarning**(ostream&) **setOut**(ostream&)	redirect error, warning and output channels for solver

Environment and model:

- **IloEnv** : create a common environment (build + solve model)

end()	destroy environment (replace destructor) as well as ALL objects tied to this environment
setError(ostream&) **setWarning**(ostream&) **setOut**(ostream&)	redirect error, warning and output channels for environment
getMemoryUsage()	heap usage (in bytes)
getTotalMemoryUsage()	allocated heap (in bytes)

ex.: ***IloEnv*** *env;*

- **IloModel** : create LP, QP, MILP, MIQP model (with variables, constraints, etc.)

add(const IloExtractableArray &X) **add**(const IloExtractable &X)	add element or array X to model
remove(const IloExtractableArray &X) **remove**(const IloExtractable &X)	remove element or array X from model
inherited methods	*see IloExtractable*

==> you should add to model : variables, constraints, objective

ex.: ***IloModel*** *model(env);*
model.add(obj);

Objectif:

- **IloObjective**: define objective function

setConstant(IloNum)	sets constant term of objective
setLinearCoef(*variable, value*) **setLinearCoefs**(*variables, values*)	modify one or multiple objective coefficients
setExpr(*expression*)	sets expression of objective
setSense(*IloObjective::Sense*)	sets sense (*min*/*max*) of the objective
inherited methods	*see IloExtractable*

ex.: ilobendersatsp.cpp

IloObjective *obj(env);*
obj.setSense(IloObjective::Minimize);
obj.setExpr(exp);

N.B.: *IloObjective::Sense* = IloObjective::Minimize *or* IloObjective::Maximize

- **IloMinimize**, **IloMaximize** : sets optimization direction (*min* / *max*)

ex.: ***IloObjective*** *obj* = ***IloMinimize****(env);*

Variables:

- **IloNumColumn** : populate variables adding coefficients one by one

clear()	reinitialize object
operator +=()	add coefficients to the variable
end()	destroy object

N.B.: USE *end()* METHOD ONCE VARIABLE HAS BEEN DEFINED IN ORDER TO PREVENT MEMORY LEAKS

- **IloNumVar, IloIntVar, IloBoolVar** : declare numeric, integer or binary variables by default

setLB(*value*) **setUB**(*value*) **setBounds**(*value, value*)	modify variable bounds (lower, upper or both)
inherited methods	*see IloExtractable*

ex.: ilodiet.cpp

```
IloNumVarArray buy(env);
for (j = 0; j < n; j++) {
    IloNumColumn col = cost(foodCost[j]);
    for (i = 0; i < m; i++)
        col += range[i](nutrPer[i][j]);
    Buy.add(IloNumVar(col, foodMin[j], foodMax[j], type));
    col.end();
}
```

- **IloConversion**: convert numeric variables in other types

end()	end the conversion **(GET BACK TO INITIAL TYPE)**
inherited methods	*see IloExtractable*

N.B.: We cannot add 2 conversions back to back to the same variables!!!

ex.:

```
IloConversion conv(env, varX, ILOINT);
model.add(conv);
model.remove(conv);
conv.end();
```

Constraints:

- **IloExpr**: build an expression (constraint)

setConstant(IloNum)	initialize the constant term of the expression
setLinearCoef(*variable, value*) **setLinearCoefs**(*variables, values*)	change some coefficients of the expression
operator += (*variable*/*expression*/*value*)	add an element to the expression (with positive sign)
operator -= (*variable*/*expression*/*value*)	add an element to the expression (with negative sign)
inherited methods	*see IloExtractable*

N.B.: **USE *end()* METHOD ONCE VARIABLE HAS BEEN DEFINED IN ORDER TO PREVENT MEMORY LEAKS**

- **IloRange**: declare constraints

setLB(*value*) **setUB**(*value*) **setBounds**(*value, value*)	change bounds of constraint (lower, upper or both)
setLinearCoef(*variable, value*) **setLinearCoefs**(*variables, values*)	change some coefficients of constraint
setExpr(*expression*)	Set a new expression to the constraint (LHS)
inherited methods	*see IloExtractable*

ex.: facility.cpp

```
for(j = 0 ; j < nbLocations ; j++){
    IloExpr v(env);
    for(i = 0; i < nbClients; i++)
        v += supply[i][j];
    model.add(v <= capacity[j] * open[j]);
            // ou model.add(IloRange(env, 0, capacity[j] * open[j] - v, IloInfinity);
    v.end();
}
```

- **IloSum**(*variables*) : return the sum of the variables in the array

  ex.: facility.cpp

  ```
  IloArray<IloNumVarArray> supply(env, nbClients);

  for(i = 0; i < nbClients; i++)
        supply[i] = IloNumVarArray(env, nbLocations, 0, 1, ILOINT);
  for(i = 0; i < nbClients; i++)
        model.add(IloSum(supply[i]) == 1);    // ∑ (j) supply[i][j] == 1
  ```

  N.B.: IloSum can be applied only on the LAST dimension of the matrix.
  *Otherwise, we have to use **IloExpr** to build the constraint.*

- **IloScalProd**(*variables*, *values*): return the scalar product of an array of variables and an arayy of values

  ex.: facility.cpp

  ```
  IloNumArray    fixedCost(env);
  IloNumVarArray open(env, nbLocations, 0, 1, ILOINT);
  IloArray<IloNumVarArray> supply(env, nbClients);
  IloExpr obj = IloScalProd(fixedCost, open);    // ∑(j)  fixedCost[j] * open[j]

  for(i = 0; i < nbClients; i++)
        obj += IloScalProd(cost[i], supply[i]);    // ∑(i,j)  supply[i][j] * cost[i][j]
  ```

- **IloIfThen**(*condition If, condition Then*) : create « Big-M » or logic constraints

  *ex.: x <= M * y ==> **IloIfThen**(env, y == 0, x == 0);*

  ex.: foodmanufact.cpp

  If products p1 or p2 are used (more than 20), then product p3 will be used also:

  ```
  model.add(IloIfThen(env, (use[p1] >= 20) || (use[p2] >= 20),  use[p3] >= 20));
  ```

Tableaux + macro:

- **IloAdd** : macro which defines objects and add them to the model at the same time

  ex.: ilodiet.cpp

  IloObjective cost = ***IloAdd****(mod, IloMinimize(env));*
  // create objective and add it to the model
  ==> *equivalent to:*

  IloObjective cost = IloMinimize(env);
  mod.add(cost);

- **IloArray** : class template for multi-dimensions extensible array

add(X)	add X to the array
operator[](int i)	get access to the i-th element of the array
clear()	clear the array => size = 0
getSize()	returns the size of the array
end()	destroy the array and all its elements (replaces destructor)

ex.: facility.cpp

typedef ***IloArray<IloNumArray>*** *FloatMatrix;* // 2D matrix of numbers
typedef ***IloArray<IloNumVarArray>*** *NumVarMatrix;* // 2D matrix of variables

- **IloExtractableArray** : class for using extensible arrays to define one or multi-dimensional arrays for variables/constraints

end()	destroy array and all its elements (replaces destructor)
removeFromAll()	remove elements of the array from all other objects where they were referenced
setNames(const char*)	give name to the elements of the array

- **IloNumArray, IloIntArray, IloBoolArray** : extensible arrays for numbers

contains(*value*)	look for a specific value in the array
inherited methods	*see IloExtractableArray et IloArray*

- **IloNumVarArray, IloIntVarArray, IloBoolVarArray** : declare numeric, integer or binary arrays of variables

add(*variable*) **add**(*variables*)	add variables to the array
setBounds(*values, values*)	change bounds of variables in the array
inherited methods	*see IloExtractableArray et IloArray*

ex.: facility.cpp

```
typedef IloArray<IloNumVarArray> NumVarMatrix;
                                        // 2D matrix of variables

NumVarMatrix supply(env, nbClients);
for(i = 0; i < nbClients; i++)
      supply[i] = IloNumVarArray(env, nbLocations, 0, 1, ILOINT);
```

ex.: cutstock.cpp

```
IloNumArray     newPatt(env, nWdth);
IloNumVarArray  Use(env, nWdth, 0.0, IloInfinity, ILOINT);
patSolver.getValues(newPatt, Use);
```

- **IloRangeArray**: declare extensible arrays of constraints

add(*constraint*) **add**(*constraints*)	add constraints to the array
setBounds(*values, values*)	Change bounds of constraints in the array
inherited methods	*see IloExtractableArray et IloArray*

ex.:
```
IloRangeArray  tab(env);
   tab.add(IloRange(env, 0.0, 100.0));
IloRangeArray range (env, nutrMin, nutrMax);    ===>  ilodiet.cpp
```

Solver class:

- **IloCplex** : solving a model with CPLEX

LP + MIP	**solve**()	solve the model (LP relaxation followed by MIP if needed)
	getCplexStatus()	returns optimization status
	exportModel(char*)	Export model to a file
	getObjValue()	returns objective function value
	getValue(*variable*/*expression*) **getValues**(*values, variables*)	retrieve values of variables at the end of otpimization
	setParam(*parametre, value*)	change parameter values
	tuneParam()	automatic search for better parameter values
LP	**getDual**(*constraint*), **getDuals**(*values, variables*/*expressions*)	retrieve dual values of constraints
	getReducedCost(*variable*) **getReducedCosts**(*values*, *variables*)	retrieve reduced costs of variables
	getSlack(*constraint*), **getSlacks**(*values, constraints*)	retrieve slack value associated to constraints
MIP	**getBestObjValue**()	retrieve best known bound of all the remaining open nodes
	addLazyConstraint(*constraint*) **addLazyConstraints**(*constraints*)	add LAZY constraints in a cut pool (integer solutions)
	addUserCut(*constraint*), **addUserCuts**(*constraints*)	add USER constraints in a cut pool (any non-integer node)
	solveFixed()	solve MIP while fixing variables to LP solution (get access to duals, slacks, ...)
	addMIPStart(*variables, values*)	define starting point to solve MIP
	setPriority(*variable, value*) **setPriorities**(*variables, values*)	Set priorities on variables for MIP branching
	use(IloCplex::Callback)	Usage of callbacks to change the progress of MIP B&C
	inherited methods	*see IloAlgorithm*

Useful parameters:

IloCplex::Param::	*(prefix)*
Threads	**Limit on number of threads used (default: ALL CPUs) Please use : 1**
Advance	keep the current basis or not
TimeLimit	max time for solving (in seconds)
ClockType	to specify wall clock or CPU time
Preprocessing::Presolve	apply presolve or not
Preprocessing::Symmetry	Break model symmetry (MIP)
Preprocessing::BoundStrength	Ry to fix variables (MIP)
Simplex::Tolerances::Optimality	tolerance on solution optimality
Simplex::Tolerances::Feasibility	tolerance on feasibility of variables
Simplex::Display	display simplexe optimization
Emphasis::MIP	MIP search emphasis (FEASIBLE vs OPTIMAL)
Emphasis::Numerical	numerical precision
Emphasis::Memory	keep memory usage as low as possible
RootAlgorithm	algorithm to solve root node
NodeAlgorithm	algorithm to solve other nodes
MIP::Display	display of B&C
MIP::Strategy::File	create a file to store nodes
WorkMem	limit on memory used before storing nodes
MIP::Strategy::VariableSelect	select next variable (MIP branching)
MIP::Strategy::NodeSelect	select next node (MIP branching)
MIP::Strategy::Branch	decide which branch should be taken first
MIP::Tolerances::LowerCutoff	cut branches where solution value < threshold
MIP::Tolerances::UpperCutoff	cut branches where solution value > threshold
MIP::Tolerances::MIPGap	limit on gap (%)
MIP::Tolerances::Integrality	tolerance on integrality of variables
MIP::Limits::Solutions	limit on number of integer solutions found
MIP::Limits::Nodes	limit on number of evaluated nodes
MIP::Limits::TreeMemory	limit on branching tree memory

Callbacks : (*OPTIONAL*)

==> to give users a way to change the progress of branch&cut.

- Callbacks types:

informational	retrieve information on current optimization without changing anything on the solving process
diagnostic	check the progress of optimization and eventually terminate it
control	control the search process during B&C procedure and interfere directly in the solving process

- Control callbacks:

UserCutCallback	add cuts to non-integer nodes
LazyConstraintCallback	add cuts to integer nodes
NodeCallback	choose the next node the B&C will be solving
SolveCallback	change the way the node is solved
HeuristicCallback	implement heuristic to give better incumbents
BranchCallback	choose the next nodes to be created
IncumbentCallback	check incumbent solutions and reject « false » integer solutions

- Macros to define callbacks (LAZY/USER) :

ILOLAZYCONSTRAINTCALLBACK0(name)
ILOLAZYCONSTRAINTCALLBACK1(name, type1, x1)
ILOLAZYCONSTRAINTCALLBACK2(name, type1, x1, type2, x2)
...
ILOLAZYCONSTRAINTCALLBACK7 (*name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7*)
ILOUSERCUTCALLBACK0(name)
ILOUSERCUTCALLBACK1(name, type1, x1)
ILOUSERCUTCALLBACK2(name, type1, x1, type2, x2)
...
ILOUSERCUTCALLBACK7 (*name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7*)

ex.: ***ILOUSERCUTCALLBACK3****(*CtCallback*, IloExprArray, lhs, IloNumArray, rhs, IloNum, eps)*

-> define a function of type *UserCutCallback* with name *CtCallback* receiving 3 parameters (*lhs, rhs, eps*) of types (*IloExprArray, IloNumArray, IloNum*)

*ex.: cplex.use(*CutCallback*(env, lhs, rhs, 1e-05);*

-> ask the solver to use the UserCutCallback named *CutCallback* during optimization with parameters: lhs, rhs, 1e-05

LAZY + USER	**add**(*constraint*)	add global cut to the model
	addLocal(*constraint*)	add local cut to the model
	inherited methods	*see next table*
USER	**abortCutLoop**()	get out of the cutting phase and go back to branching
	isAfterCutLoop()	returns IloTrue if callback called one last time after the last cut has been generated (IloFalse otherwise)

- inherited methods (*common to LazyConstraintCallback and UserCutCallback*)

getLB(*variable*) **getLBs**(*values, variables*) **getUB**(*variable*) **getUBs**(*values, variables*)	retrieve lower/upper bounds from variables
getNodeId()	retrieve unique ID from current node
getObjValue()	retrieve objective value of current node
getSlack(*constraint*) **getSlacks**(*constraints*)	retrieve slack values of constraints at current node
getValue(*variable*/*expression*) **getValues**(*values, variables*)	retrieve values of variables (Expression) at current node
getBestObjValue()	retrieve best bound of the remaining nodes
getIncumbentObjValue()	retrieve best incumbent found so far
getIncumbentValue (*variable*/*expression*) **getIncumbentValues** (*values, variables*)	retrieve values of variables (expressions) from the best integer solution found so far
getNnodes()	retrieve the number of evaluated nodes
getNremainingNodes()	retrieve the number of remaining nodes
hasIncumbent()	returns IloTrue if an integer solution has been found (IloFalse otherwise)
getModel()	retrieve the current model
abort()	terminate optimization
getEnv()	retrieve current environment

N.B.: Inside the user-defined function (macro), the only way to retrieve values of variables/constraints at current node is to give all needed arrays as parameters to the callback like this :

ex.: ilobendersatsp.cpp

```
ILOUSERCUTCALLBACK5(BendersUserCallback, IloArray<IloIntVarArray>, x,
                                         IloCplex, workerCplex, IloNumVarArray, v,
                                         IloNumVarArray, u, IloObjective, workerObj)
{
  // Skip the separation if not at the end of the cut loop

  if ( !isAfterCutLoop() )      return;

  IloInt i;
  IloEnv masterEnv = getEnv();
  IloInt numNodes = x.getSize();

  // Get the current x solution

  IloArray<IloNumArray> xSol(masterEnv, numNodes);
  for (i = 0; i < numNodes; ++i) {
    xSol[i] = IloNumArray(masterEnv);
    getValues(xSol[i], x[i]);
  }

  // Benders' cut separation

  IloExpr cutLhs(masterEnv);        // expression de la coupe
  IloNum cutRhs;

  IloBool sepStat = separate(x, xSol, workerCplex, v, u, workerObj, cutLhs, cutRhs);
  if ( sepStat ) {
    add(cutLhs >= cutRhs).end();
     // cree la coupe, l'ajoute au modele et detruit l'objet temporaire cree
  }

  // Free memory

  cutLhs.end();
  for (i = 0; i < numNodes; ++i)
    xSol[i].end();
  xSol.end();
  return;
}
```