CPLEX Tutorial Handout

What Is ILOG CPLEX?

ILOG CPLEX is a tool for solving linear optimization problems, commonly referred to as Linear Programming (LP) problems, of the form:

Maximize (or Minimize) $c_1x_1 + c_2x_2 + ... + c_nx_n$

subject to
$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n \sim b_1$$
$$a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n \sim b_2$$
$$...$$
$$a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n \sim b_m$$

with these bounds
$$l_1 \leq x_1 \leq u_1$$
$$...$$
$$l_n \leq x_n \leq u_n$$

where ~ can be $\leq$, $\geq$, or =, and the upper bounds u_i and lower bounds l_i may be positive infinity, negative infinity, or any real number.

The elements of data you provide as input for this LP are:

Objective function coefficients $c_1, c_2, ..., c_n$

Constraint coefficients
$$a_{11}, a_{21}, ..., a_{n1}$$
$$...$$
$$a_{m1}, a_{m2}, ..., a_{mn}$$

Right-hand sides $b_1, b_2, ..., b_m$

Upper and lower bounds $u_1, u_2, ..., u_n$ *and* $l_1, l_2, ..., l_n$

The optimal solution that ILOG CPLEX computes and returns is: Variables $x_1, x_2, ..., x_n$

ILOG CPLEX also can solve several extensions to LP:

- Network Flow problems, a special case of LP that CPLEX can solve much faster by exploiting the problem structure.
- Quadratic Programming (QP) problems, where the LP objective function is expanded to include quadratic terms.
- Mixed Integer Programming (MIP) problems, where any or all of the LP or QP variables are further restricted to take integer values in the optimal solution and where MIP itself is extended to include constructs like Special Ordered Sets (SOS) and semi-continuous variables.

ILOG CPLEX Components

CPLEX comes in three forms to meet a wide range of users' needs:

- The **CPLEX Interactive Optimizer** is an executable program that can read a problem interactively or from files in certain standard formats, solve the problem, and deliver the solution interactively or into text files. The program consists of the file cplex.exe on Windows platforms or cplex on UNIX platforms.
- **Concert Technology** is a set of C++, Java, and .NET class libraries offering an API that includes modeling facilities to allow the programmer to embed CPLEX optimizers in C++, Java, or .NET applications. The following table lists the files that contain the libraries.

	Microsoft Windows	**UNIX**
C++	ilocplex.lib concert.lib	libilocplex.a libconcert.a
Java	cplex.jar	cplex.jar
C#.NET	ILOG.CPLEX.dll ILOG.CONCERT.dll	

- The **CPLEX Callable Library** is a C library that allows the programmer to embed ILOG CPLEX optimizers in applications written in C, Visual Basic, FORTRAN, or any other language that can call C functions.The library is provided in files cplex.lib and cplex.dll on Windows platforms, and in libcplex.a, libcplex.so, and libcplex.sl on UNIX platforms.

Solving an LP with ILOG CPLEX

The problem to be solved is:

Maximize $x1 + 2x2 + 3x3$

subject to $-x1 + x2 + x3 \leq 20$
$x1 - 3x2 + x3 \leq 30$

with these bounds $0 \leq x1 \leq 40$
$0 \leq x2 \leq +\infty$
$0 \leq x3 \leq +\infty$

Using the Interactive Optimizer

The following sample is screen output from a CPLEX Interactive Optimizer session where the model of an example is entered and solved. CPLEX> indicates the CPLEX prompt, and text following this prompt is user input.

```
Welcome to CPLEX Interactive Optimizer 9.0.0
  with Simplex, Mixed Integer & Barrier Optimizers
Copyright (c) ILOG 1997-2003
CPLEX is a registered trademark of ILOG

Type 'help' for a list of available commands.
Type 'help' followed by a command name for more
information on commands.

CPLEX> enter example
Enter new problem ['end' on a separate line terminates]:
maximize  x1 + 2 x2 + 3 x3
```

```
subject to -x1 +  x2 + x3 <= 20
        x1 - 3 x2 + x3 <=30
bounds
0 <= x1 <= 40
0 <= x2
0 <= x3
end
CPLEX> optimize
Tried aggregator 1 time.
No LP presolve or aggregator reductions.
Presolve time =   0.00 sec.

Iteration log . . .
Iteration:    1  Dual infeasibility =         0.000000
Iteration:    2  Dual objective    =       202.500000

Dual simplex - Optimal:  Objective =  2.0250000000e+002
Solution time =   0.01 sec.  Iterations = 2 (1)

CPLEX> display solution variables x1-x3
Variable Name        Solution Value
x1                   40.000000
x2                   17.500000
x3                   42.500000
CPLEX> quit
```

Concert Technology for Java Users

- Creating a model
- Solving that model
- Querying results after solving
- Handling error conditions

ILOG Concert Technology allows your application to call ILOG CPLEX directly. This Java interface supplies a rich means for you to use Java objects to build your optimization model. The IloCplex class implements the ILOG Concert Technology interface for creating variables and constraints. It also provides functionality for solving Mathematical Programing (MP) problems and accessing solution information.

Compiling ILOG CPLEX Applications in ILOG Concert Technology

For this, you add the `cplex.jar` file to your classpath. This is most easily done by passing the command-line option to the Java compiler javac:

```
-classpath <path_to_cplex.jar>
```

In the java command line, the following should be added.

```
-Djava.library.path=<path_to_shared_library>
```

See the `makefile` at http://studentweb.engr.utexas.edu/wangym .

The Anatomy of an ILOG Concert Technology Application

To use the ILOG CPLEX Java interfaces, you need to import the appropriate packages into your application. This is done with the lines:

```
import ilog.concert.*;
import ilog.cplex.*;
```

As for every Java application, an ILOG CPLEX application is implemented as a method of a class. In this discussion, the method will be the static main method. The first task is to create an IloCplex object. It is used to create all the modeling objects needed to represent the model. For example, an integer variable with bounds 0 and 10 is created by calling cplex.intVar(0, 10), where cplex is the IloCplex object.
Since Java error handling in ILOG CPLEX uses exceptions, you should include the ILOG Concert Technology part of an application in a try/catch statement. All the exceptions thrown by any ILOG Concert Technology method are derived from IloException. Thus IloException should be caught in the catch statement.
In summary, here is the structure of a Java application that calls ILOG CPLEX:

```
import ilog.concert.*;
import ilog.cplex.*;
static public class Application {
  static public main(String[] args) {
    try {
      IloCplex cplex = new IloCplex();
      // create model and solve it
    } catch (IloException e) {
       System.err.println("Concert exception caught: " + e);
    }
   }
 }
```

Create the Model

The IloCplex object provides the functionality to create an optimization model that can be solved with IloCplex. The interface functions for doing so are defined by the ILOG Concert Technology interface IloModeler and its extension IloMPModeler. These interfaces define the constructor functions for modeling objects of the following types, which can be used with IloCplex:

IloNumVar	modeling variables
IloRange	ranged constraints of the type lb <= expr <= ub
IloObjective	optimization objective
IloNumExpr	expression using variables

Modeling variables are represented by objects implementing the IloNumVar interface defined by ILOG Concert Technology. Here is how to create three continuous variables, all with bounds 0 and 100:

```
IloNumVar[] x = cplex.numVarArray(3, 0.0, 100.0);
```

There is a wealth of other functions for creating arrays or individual modeling variables. The documentation for IloModeler and IloMPModeler will give you the complete list.

Modeling variables are typically used to build expressions, of type IloNumExpr, for use in constraints or the objective function of an optimization model. For example the expression:

```
x[0] + 2*x[1] + 3*x[2]
```

can be created like this:

```
IloNumExpr expr = cplex.sum(x[0], cplex.prod(2.0, x[1]),
                            cplex.prod(3.0, x[2]));
```

Another way of creating an object representing the same expression is to use an IloLinearNumExpr expression. Here is how:

```
IloLinearNumExpr expr = cplex.linearNumExpr();
expr.addTerm(1.0, x[0]);
expr.addTerm(2.0, x[1]);
expr.addTerm(3.0, x[2]);
```

The advantage of using IloLinearNumExpr over the first way is that you can more easily build up your linear expression in a loop, which is what is typically needed in more complex applications. Interface IloLinearNumExpr is an extension of IloNumExpr, and thus can be used anywhere an expression can be used. As mentioned before, expressions can be used to create constraints or an objective function for a model. Here is how to create a minimization objective for the above expression:

```
IloObjective obj = cplex.minimize(expr);
```

In addition to creating an objective, IloCplex must be instructed to use it in the model it solves. This is done by adding the objective to IloCplex via:

```
cplex.add(obj);
```

Every modeling object that is to be used in a model must be added to the IloCplex object. The variables need not be explicitly added as they are treated implicitly when used in the expression of the objective. More generally, every modeling object that is referenced by another modeling object which itself has been added to IloCplex, is implicitly added to IloCplex as well.
There is a shortcut notation for creating and adding the objective to IloCplex:

```
cplex.addMinimize(expr);
```

Since the objective is not otherwise accessed, it does not need to be stored in the variable obj.
Adding constraints to the model is just as easy. For example, the constraint
-x[0] + x[1] + x[2] <= 20.0
can be added by calling:

```
cplex.addLe(cplex.sum(cplex.negative(x[0]), x[1], x[2]), 20);
```

Again, many methods are provided for adding other constraint types, including equality constraints, greater than or equal to constraints, and ranged constraints. Internally, they are all represented as IloRange objects with appropriate choices of bounds, which is why all these methods return IloRange objects. Also, note that the expressions above could have been created in many different ways, including the use of IloLinearNumExpr.

Solve the Model

IloCplex.solve()
IloCplex.solveRelaxed()

IloCplex.getStatus.
The returned value tells you what ILOG CPLEX found out about the model: whether it found the optimal solution or only a feasible solution, whether it proved the model to be unbounded or infeasible, or whether

nothing at all has been determined at this point. Even more detailed information about the termination of the solver call is available through the method IloCplex.getCplexStatus.

Query the Results

If the solve method succeeded in finding a solution, you will then want to access that solution. The objective value of that solution can be queried using a statement like this:

```
double objval = cplex.getObjValue();
```

Similarly, solution values for all the variables in the array x can be queried by calling:

```
double[] xval = cplex.getValues(x);
```

More solution information can be queried from IloCplex, including slacks and, depending on the algorithm that was applied for solving the model, duals, reduced cost information, and basis information.

Building and Solving a Small LP Model in Java

The example LPex1.java, part of the standard distribution of ILOG CPLEX, is a program that builds a specific small LP model and then solves it. This example follows the general structure found in many ILOG CPLEX Concert Technology applications, and demonstrates three main ways to construct a model:

- *Modeling by Rows*;
- *Modeling by Columns*;

Example LPex1.java

Maximize	$x1 + 2x2 + 3x3$
subject to	$-x1 + x2 + x3 \leq 20$
	$x1 - 3x2 + x3 \leq 30$
with these bounds	$0 \leq x1 \leq 40$
	$0 \leq x2 \leq +\infty$
	$0 \leq x3 \leq +\infty$

Program for building and solving the example

```
import ilog.concert.*;
import ilog.cplex.*;

public class Example {
 public static void main(String[] args) {
  try {
   IloCplex cplex = new IloCplex();

   double[]   lb = {0.0, 0.0, 0.0};
   double[]   ub = {40.0, Double.MAX_VALUE, Double.MAX_VALUE};
   IloNumVar[] x  = cplex.numVarArray(3, lb, ub);

   double[] objvals = {1.0, 2.0, 3.0};
```

```
    cplex.addMaximize(cplex.scalProd(x, objvals));

    cplex.addLe(cplex.sum(cplex.prod(-1.0, x[0]),
                  cplex.prod( 1.0, x[1]),
                  cplex.prod( 1.0, x[2])), 20.0);
    cplex.addLe(cplex.sum(cplex.prod( 1.0, x[0]),
                  cplex.prod(-3.0, x[1]),
                  cplex.prod( 1.0, x[2])), 30.0);

    if ( cplex.solve() ) {
      cplex.out().println("Solution status = " + cplex.getStatus());
      cplex.out().println("Solution value  = " + cplex.getObjValue());

      double[] val = cplex.getValues(x);
      int ncols = cplex.getNcols();
      for (int j = 0; j < ncols; ++j)
        cplex.out().println("Column: " + j + " Value = " + val[j]);
    }
    cplex.end();
   }
   catch (IloException e) {
    System.err.println("Concert exception '" + e + "' caught");
   }
  }
}
```

Compile and run

```
% javac example.java
% java example
```

or if you use makefile

```
% make
```

Complete Code of LPex1.java

```
// --------------------------------------------------------------------------
// File: examples/src/LPex1.java
// Version 9.0
// --------------------------------------------------------------------------
// Copyright (C) 2001-2003 by ILOG.
// All Rights Reserved.
// Permission is expressly granted to use this example in the
// course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// LPex1.java - Entering and optimizing an LP problem
//
// Demonstrates different methods for creating a problem.  The user has to
// choose the method on the command line:
```

```
//
//   java LPex1 -r    generates the problem by adding constraints
//   java LPex1 -c    generates the problem by adding variables
//   java LPex1 -n    generates the problem by adding expressions
//

import ilog.concert.*;
import ilog.cplex.*;


public class LPex1 {
  static void usage() {
    System.out.println("usage:  LPex1 <option>");
    System.out.println("options:     -r  build model row by row");
    System.out.println("options:     -c  build model column by column");
    System.out.println("options:     -n  build model nonzero by nonzero");
  }

  public static void main(String[] args) {
    if ( args.length != 1 || args[0].charAt(0) != `-' ) {
      usage();
      return;
    }

    try {
      // Create the modeler/solver object
      IloCplex cplex = new IloCplex();

      IloNumVar[][] var = new IloNumVar[1][];
      IloRange[][]  rng = new IloRange[1][];

     // Evaluate command line option and call appropriate populate method.
     // The created ranges and variables are returned as element 0 of arrays
     // var and rng.
      switch ( args[0].charAt(1) ) {
      case `r': populateByRow(cplex, var, rng);
             break;
      case `c': populateByColumn(cplex, var, rng);
             break;
      case `n': populateByNonzero(cplex, var, rng);
             break;
      default: usage();
             return;
      }

      // write model to file
      cplex.exportModel("lpex1.lp");

      // solve the model and display the solution if one was found
      if ( cplex.solve() ) {
        double[] x    = cplex.getValues(var[0]);
        double[] dj   = cplex.getReducedCosts(var[0]);
        double[] pi   = cplex.getDuals(rng[0]);
        double[] slack = cplex.getSlacks(rng[0]);

        cplex.output().println("Solution status = " + cplex.getStatus());
        cplex.output().println("Solution value  = " + cplex.getObjValue());

        int ncols = cplex.getNcols();
        for (int j = 0; j < ncols; ++j) {
```

```
        cplex.output().println("Column: " + j +
                               " Value = " + x[j] +
                               " Reduced cost = " + dj[j]);
      }

      int nrows = cplex.getNrows();
      for (int i = 0; i < nrows; ++i) {
        cplex.output().println("Row  : " + i +
                               " Slack = " + slack[i] +
                               " Pi = " + pi[i]);
      }
    }
    cplex.end();
  }
  catch (IloException e) {
    System.err.println("Concert exception `" + e + "` caught");
  }
}


// The following methods all populate the problem with data for the following
// linear program:
//
//   Maximize
//    x1 + 2 x2 + 3 x3
//   Subject To
//    - x1 + x2 + x3 <= 20
//    x1 - 3 x2 + x3 <= 30
//   Bounds
//    0 <= x1 <= 40
//   End
//
// using the IloMPModeler API

static void populateByRow(IloMPModeler model,
                          IloNumVar[][] var,
                          IloRange[][] rng) throws IloException {
  double[]   lb = {0.0, 0.0, 0.0};
  double[]   ub = {40.0, Double.MAX_VALUE, Double.MAX_VALUE};
  IloNumVar[] x  = model.numVarArray(3, lb, ub);
  var[0] = x;

  double[] objvals = {1.0, 2.0, 3.0};
  model.addMaximize(model.scalProd(x, objvals));

  rng[0] = new IloRange[2];
  rng[0][0] = model.addLe(model.sum(model.prod(-1.0, x[0]),
                                    model.prod( 1.0, x[1]),
                                    model.prod( 1.0, x[2])), 20.0);
  rng[0][1] = model.addLe(model.sum(model.prod( 1.0, x[0]),
                                    model.prod(-3.0, x[1]),
                                    model.prod( 1.0, x[2])), 30.0);
}

static void populateByColumn(IloMPModeler model,
                             IloNumVar[][] var,
                             IloRange[][] rng) throws IloException {
  IloObjective obj = model.addMaximize();

  rng[0] = new IloRange[2];
  rng[0][0] = model.addRange(-Double.MAX_VALUE, 20.0);
```

```
    rng[0][1] = model.addRange(-Double.MAX_VALUE, 30.0);

    IloRange r0 = rng[0][0];
    IloRange r1 = rng[0][1];

    var[0] = new IloNumVar[3];
    var[0][0] = model.numVar(model.column(obj,  1.0).and(
                    model.column(r0,  -1.0).and(
                    model.column(r1,   1.0))),
                    0.0, 40.0);
    var[0][1] = model.numVar(model.column(obj,  2.0).and(
                    model.column(r0,   1.0).and(
                    model.column(r1,  -3.0))),
                    0.0, Double.MAX_VALUE);
    var[0][2] = model.numVar(model.column(obj,  3.0).and(
                    model.column(r0,   1.0).and(
                    model.column(r1,   1.0))),
                    0.0, Double.MAX_VALUE);
  }

  static void populateByNonzero(IloMPModeler model,
                      IloNumVar[][] var,
                      IloRange[][] rng) throws IloException {
    double[]   lb = {0.0, 0.0, 0.0};
    double[]   ub = {40.0, Double.MAX_VALUE, Double.MAX_VALUE};
    IloNumVar[] x  = model.numVarArray(3, lb, ub);
    var[0] = x;

    double[] objvals = {1.0, 2.0, 3.0};
    model.add(model.maximize(model.scalProd(x, objvals)));

    rng[0] = new IloRange[2];
    rng[0][0] = model.addRange(-Double.MAX_VALUE, 20.0);
    rng[0][1] = model.addRange(-Double.MAX_VALUE, 30.0);

    rng[0][0].setExpr(model.sum(model.prod(-1.0, x[0]),
                        model.prod( 1.0, x[1]),
                        model.prod( 1.0, x[2])));
    rng[0][1].setExpr(model.sum(model.prod( 1.0, x[0]),
                        model.prod(-3.0, x[1]),
                        model.prod( 1.0, x[2])));
  }
}
```