ILOG CPLEX 7.5

User's Manual

November 2001

C O N T E N T S

Table of Contents

List of Figures

List of Tables

P R E F A C E

Meet ILOG CPLEX

ILOG CPLEX offers C and C++ libraries that solve linear programming (LP) problems and related problems. Specifically, it solves linearly constrained optimization problems where the objective to be optimized can be expressed as a linear function or a convex quadratic function. In the linear case, the variables in the model may be declared as continuous or further constrained to take only integer values.

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 libraries that offers an API that includes modeling facilities to allow the programmer to embed CPLEX optimizers in a C++ application. The library is provided in files `ilocplex.lib` and `concert.lib` on Windows platforms and in `libilocplex.a` and `libconcert.a` on UNIX platforms, and makes use of the Callable Library (described next).

◆ The **CPLEX Callable Library** is a C library that allows the programmer to embed CPLEX optimizers in applications written in C, Visual Basic, Java, and Fortran. The library is provided in files `cplex70.lib` and `cplex70.dll` on Windows platforms and in `libcplex.a` on UNIX platforms.

In this manual, the phrase **CPLEX Component Libraries** is used when referring equally to either of these two libraries. While both libraries are callable, the term CPLEX Callable Library as used here refers specifically to the C library.

This preface introduces ILOG CPLEX, version 7.1. It includes the following sections:

- What Is ILOG CPLEX?
- What You Need to Know
- In This Manual
- Notation in This Manual
- Related Documentation
- For More Information

What Is ILOG CPLEX?

ILOG CPLEX is a tool for solving, first of all, *linear* optimization problems. Such problems are conventionally written like this:

Minimize (or maximize)	$c_1x_1 + c_2x_2 + \ldots + c_nx_n$		
subject to	$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n$	$\sim$	b_1
	$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n$	$\sim$	b_2
	$\ldots$		
	$a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n$	$\sim$	b_m
with these bounds	$l_1 \leq x_1 \leq u_1, \ldots, l_n \leq x_n \leq u_n$		

where the relation ~ may be greater than or equal to, less than or equal to, or simply equal to, and the upper bounds u_i and lower bounds l_i may be positive infinity, negative infinity, or any real number.

When a linear optimization problem is stated in that conventional form, we customarily refer to its coefficients and values by these terms:

objective function coefficients	$c_1,$	$\ldots,$	c_n
constraint coefficients	$a_{11},$	$\ldots,$	a_{mn}
right-hand side	$b_1,$	$\ldots,$	b_m
upper bounds	$u_1,$	$\ldots,$	u_n
lower bounds	$l_1,$	$\ldots,$	l_n
variables or unknowns	$x_1,$	$\ldots,$	x_n

In the most basic linear optimization problem, the variables of the objective function are *continuous* in the mathematical sense, with no gaps between real values. To solve such linear programming problems, ILOG CPLEX implements optimizers based on the simplex algorithms (both primal and dual simplex) as well as primal-dual logarithmic barrier algorithms. These alternatives are explained more fully in Chapter 4, *Solving Linear Programming Problems.*

ILOG CPLEX is also a tool for solving linear programming problems in which some or all of the variables must assume *integer* values in the solution. Such problems are known as *mixed integer programs* or MIPs because they may combine continuous and discrete (for example, integer) variables in the objective function and constraints. Within the category of mixed integer programs, we distinguish two kinds of discrete integer variables: if the integer values of the discrete variables must be either 0 (zero) or 1 (one), then we refer to them as *binary*; if the integer values are not restricted in that way, we refer to them as general integer variables. This manual explains more about the separately licensed ILOG CPLEX Mixed Integer Optimizer in Chapter 5, *Solving Mixed Integer Programming Problems.*

ILOG CPLEX can also handle certain problems in which the objective function is not linear but *quadratic*. (The constraints in such a problem are still linear.) Such problems are known as *quadratic programs* or QPs. Chapter 7, *Solving Quadratic Programming Problems* covers those kinds of problems.

ILOG CPLEX also offers a Network Optimizer aimed at a special class of linear problem with network structures. ILOG CPLEX can optimize such problems as ordinary linear programs, but if ILOG CPLEX can extract all or part of the problem as a network, then ILOG CPLEX will apply its more efficient Network Optimizer to that part of your problem and use the partial solution it finds there to construct an advanced starting point to optimize the rest of the problem. Chapter 6, *Solving Network-Flow Problems* offers more detail about how the CPLEX Network Optimizer works.

What You Need to Know

In order to use ILOG CPLEX effectively, you need to be familiar with your operating system, whether UNIX or Windows.

In this manual, we assume you are familiar with the concepts of mathematical programming, particularly linear programming. In case those concepts are new to you, the bibliography on page 25 in this preface indicates references to help you there.

This manual also assumes you already know how to create and manage files. In addition, if you are building an application that uses the Component Libraries, this manual assumes that you know how to compile, link, and execute programs written in a high-level language. The Callable Library is written in the C programming language, while Concert Technology is written in C++. This manual also assumes that you already know how to program in the appropriate language and that you will consult a programming guide when you have questions in that area.

In This Manual

Chapter 1, *Using ILOG CPLEX Concert Technology Library* introduces the Concert Technology Library. It provides an overview of the design of the library, explains modeling techniques, and offers an example of programming with the Concert Technology Library. It also provides information on controlling parameters.

Chapter 2, *Using the ILOG CPLEX Callable Library* introduces the ILOG CPLEX Callable Library. It sketches the architecture of the product, explains the relation between the Interactive Optimizer and the Callable Library, and offers an example of programming with the Callable Library. It also provides an overview about the parameters you control in ILOG CPLEX, outlines the callable routines controlling parameters, and explains the `set` command.

Chapter 3, *Further Programming Considerations* provides tips on developing applications with CPLEX, suggests ways to debug your applications built around CPLEX, and provides a checklist to help avoid common programming errors.

Chapter 4, *Solving Linear Programming Problems* goes deeper into aspects of linear programming with ILOG CPLEX. It explains how to tune performance, how to diagnose infeasibility in a model, and how to use the primal-dual logarithmic barrier algorithm implemented in the ILOG CPLEX Barrier Optimizer on large, sparse linear programming problems. It also offers an example showing you how to start optimizing from an advanced basis.

Chapter 5, *Solving Mixed Integer Programming Problems* shows you how to handle MIPs. It particularly emphasizes performance tuning and offers a series of examples.

Chapter 6, *Solving Network-Flow Problems* describes how to use the CPLEX Network Optimizer on linear programming problems based on a network model.

Chapter 7, *Solving Quadratic Programming Problems* takes up programming problems in which the objective function may be quadratic. It, too, includes examples.

Chapter 8, *More About Using ILOG CPLEX* includes several sections on working with important aspects of the ILOG Component Libraries. Information is provided on:

- Managing Input & Output explains how to enter mathematical programs efficiently and how to generate meaningful output from your ILOG CPLEX applications. It also lists the available file formats for entering data into ILOG CPLEX and writing bases and solutions from ILOG CPLEX.
- Using Query Routines tells how to access information about the model you currently have in memory through query routines of the Callable Library.

- Using Callbacks shows how to use callbacks.
- Using Parallel Optimizers explains how to exploit parallel optimizers in case your hardware supports parallel architecture.

Appendix A, *Interactive Optimizer Commands* lists the commands available in the ILOG CPLEX Interactive Optimizer with cross-references to examples of their use in this manual. It also provides an overview about controlling parameters with the Interactive Optimizer.

Examples On-Line

For the examples that we explain in the manual, we'll also show you the complete code for the solution, so that you can see exactly how CPLEX fits into your own applications. In case you prefer to study code on-line, you'll also find the complete code for these examples in a subdirectory of the standard distribution of CPLEX.

The following table describes all the examples in this manual and indicates where to find them, both on-line and in the manual:

Example	Source File	In This Manual
dietary optimization: building a model by rows (constraints) or by columns (variables), solving with `IloCplex`	`ilodiet.cpp`	*Example: Dietary Optimization* on page 45
dietary optimization: building a model by rows (constraints) or by columns (variables), solving with the Callable Library	`diet.c`	*Example: Dietary Optimization* on page 72
linear programming: starting from an advanced basis	`ilolpex6.cpp` `lpex6.c`	*Example ilolpex6.cpp* on page 121 *Example lpex6.c* on page 123
mixed integer programming: optimizing a basic MIP	`ilomipex1.cpp` `mipex1.c`	*Complete Program: ilomipex1.cpp* on page 189 *Complete Program: mipex1.c* on page 190
mixed integer programming: reading a MIP from a formatted file	`ilomipex2.cpp` `mipex2.c`	*Example: ilomipex2.cpp* on page 199 *Example: mipex2.c* on page 201
mixed integer programming: using special ordered sets (SOS) and priority orders	`ilomipex3.cpp` `mipex3.c`	*Example: ilomipex3.cpp* on page 205 *Example: mipex3.c* on page 207
network optimization: using the Callable Library	`netex1.c`	*Complete Program: netex1.c* on page 226
network optimization: relaxing a network flow to an LP	`netex2.c`	*Complete Program: netex2.c* on page 233
quadratic programming: maximizing a QP	`qpex1.c`	*Complete Program: qpex1.c* on page 249

Example	Source File	In This Manual
quadratic programming: reading a QP from a formatted file	`qpex2.c`	*Complete Program: qpex2.c* on page 257
input and output: using the message handler	`lpex5.c`	*Complete Program: lpex5.c* on page 273
using query routines	`ilolpex7.cpp` `lpex7.c`	*Complete Program: ilolpex7.cpp* on page 284 *Complete Program: lpex7.c* on page 286
using callbacks	`ilolpex4.c` `lpex4.c`	*Complete Program: ilolpex4.cpp* on page 299 *Complete Program: lpex4.c* on page 304

Notation in This Manual

Like the reference manual, this manual uses the following conventions:

- Important ideas are *italicized* the first time they appear.
- The names of C routines and parameters in the ILOG CPLEX Callable Library begin with `CPX`; the names of C++ classes in the CPLEX Concert Technology Library begin with `Ilo`; and both appear in `this typeface`, for example: `CPXcopyobjnames()` or `IloCplex`.
- Text that is entered at the keyboard or displayed on the screen and commands and their options available through the Interactive Optimizer appear in `this typeface`, for example, `set preprocessing aggregator n`.
- Values that you must fill in (for example, the value to set a parameter) also appear in the same typeface as the command but slanted to indicate you must supply an appropriate value; for example, `set simplex refactor` *`i`* indicates that you must fill in a value for *`i`*.
- Matrices are denoted in two ways:
  - In printed material where superscripts and bold type are available, we denote the product of A and its transpose like this: $\boldsymbol{AA^T}$. The superscript T indicates the matrix transpose.
  - In computer-generated samples, such as log files, where only ASCII characters are available, we denote the product of A and its transpose like this: `A*A'`. The asterisk (*) indicates matrix multiplication, and the prime (') indicates the matrix transpose.

Related Documentation

In addition to this manual of examples, which is intended to show you how to make ILOG CPLEX work for you, the standard distribution of ILOG CPLEX comes with *Get Started with ILOG CPLEX*, the *ILOG CPLEX Reference Manual*, and the *ILOG Concert Technology Documentation Kit*. All ILOG documentation is available in an on-line version in HTML (hypertext mark-up language). It is delivered with the standard distribution of the product and accessible through conventional HTML browsers.

We strongly recommend that you begin your acquaintance with ILOG CPLEX through the introductory manual, *Getting Started with ILOG CPLEX*, which includes tutorials for the Interactive Optimizer, the Concert Technology Library, and the Callable Library. These tutorials provide a stepping-stone toward the examples in this manual.

The *ILOG CPLEX Reference Manual* documents the Callable Library routines and their arguments, the Concert Technology classes, methods, and functions, and the commands and options of the Interactive Optimizer. The *Reference Manual* also contains a table of parameters that can be modified by parameter routines, a list of error messages, and details about file formats. Consult the *Reference Manual*, whether printed or on-line, for authoritative documentation of the Component Libraries and Interactive Optimizer.

The *ILOG Concert Technology Documentation Kit* includes the *ILOG Concert Technology Reference Manual*, which documents the classes, methods, and functions of the Concert Technology library; the *ILOG Concert Technology User's Manual*, which provides examples that show how to use Concert Technology to model problems; the *ILOG Concert Technology Hybrid Optimizers User's Guide & Reference*, which documents the class `IloLinConstraint` and shows how to use ILOG's main algorithm classes, `IloSolver` and `IloCplex` in cooperation; and the *ILOG Concert Technology Migration Guide*, which shows how to translate applications created in previous versions of ILOG products to Concert Technology.

For More Information

ILOG offers technical support and comprehensive Web sites for its products.

Technical Support

For technical support of ILOG CPLEX, you should contact your local distributor, or, if you are a direct ILOG customer, contact the nearest regional office:

Region	E-mail	Telephone	Fax
France	cplex-support@ilog.fr	0 800 09 27 91 (numéro vert) +33 (0)1 49 08 35 62	+33 (0)1 49 08 35 10
Germany	cplex-support@ilog.de	+49 6172 40 60 33	+49 6172 40 60 10
Spain	cplex-support@ilog.es	+34 91 710 2480	+34 91 372 9976
United Kingdom	cplex-support@ilog.co.uk	+44 (0)1344 661600	+44 (0)1344 661601
Other European countries	cplex-support@ilog.fr	+33 (0)1 49 08 35 62	+33 (0)1 49 08 35 10
Japan	cplex-support@ilog.co.jp	+81 3 5211 5770	+81 3 5211 5771
Singapore	cplex-support@ilog.com.sg	+65 773 06 26	+65 773 04 39
USA	cplex-support@ilog.com	1-877-ILOG-TECH 1-877-456-4832 (toll free) or 1-650-567-8080	+1 650 567 8001

We encourage you to use e-mail for faster, better service.

Web Site

The CPLEX Web site at `http://www.ilog.com/products/cplex/` offers product descriptions, press releases, and contact information. It lists services, such as training, maintenance, technical support, and outlines special programs. In addition, it links you to an `ftp` site where you can pick up examples.

The technical support pages contain FAQ (Frequently Asked/Answered Questions) and the latest patches for some of our products. Changes are posted in the product mailing list. Access to these pages is restricted to owners of an ongoing maintenance contract. The maintenance contract number and the name of the person this contract is sent to in your company will be needed for access, as explained on the login page.

All three of the following sites contain the same information, but access is localized, so we recommend that you connect to the site corresponding to your location and select the "support" page from the home page.

- The Americas: `http://www.ilog.com`
- Asia & Pacific nations: `http://www.ilog.com.sg`
- Europe, Africa, and Middle East: `http://www.ilog.fr`

On those Web pages, you will find additional information about ILOG CPLEX in technical papers that have also appeared at industrial and academic conferences.

Further Reading

In case you want to know more about optimization and mathematical or linear programming, here is a brief selection of printed resources:

Williams, H. P. *Model Building in Mathematical Programming*, 4th ed. New York: John Wiley & Sons, 1999. This textbook includes many examples of how to design mathematical models, including linear programming formulations. (How you formulate your model is at least as important as what ILOG CPLEX does with it.) It also offers a description of the branch & bound algorithm.

Nemhauser, George L. and Laurence A. Wolsey, *Integer and Combinatorial Optimization*, New York: John Wiley & Sons, 1999. A reprint of the 1988 edition. A widely cited reference about integer programming, this book explains the branch & bound algorithm in detail.

Gill, Philip E., Walter Murray, and Margaret H. Wright, *Practical Optimization*. New York: Academic Press, 1982 reprint edition. This book covers, among other topics, quadratic programming.

C H A P T E R

1

Using ILOG CPLEX Concert Technology Library

This chapter describes how to write C++ programs using the ILOG CPLEX Concert Technology Library. It includes sections on:

- The Design of CPLEX Concert Technology Library, including information on licensing and on compiling and linking your programs
- Creating an Application with CPLEX Concert Technology Library
- Modeling an Optimization Problem with Concert Technology
- Solving Concert Technology Models with `IloCplex`
- Accessing Solution Information
- Modifying a Model
- Handling Errors
- Example: Dietary Optimization

The Design of CPLEX Concert Technology Library

Figure 1.1 shows a program using the CPLEX Concert Technology Library to solve optimization problems. The optimization part of the user's application program is captured in a set of interacting C++ objects that the application creates and controls. These objects can be divided into two categories:

1. **Modeling objects** are used to define the optimization problem. Generally an application creates several modeling objects to specify the optimization problems. Those objects are grouped into an `IloModel` object representing the complete optimization problem.
2. **`IloCplex` objects** are used to solve models created with the modeling objects. An `IloCplex` object reads a model and extracts its data to the appropriate representation for the CPLEX optimizer. Then the `IloCplex` object is ready to solve the model it extracted and be queried for solution information.

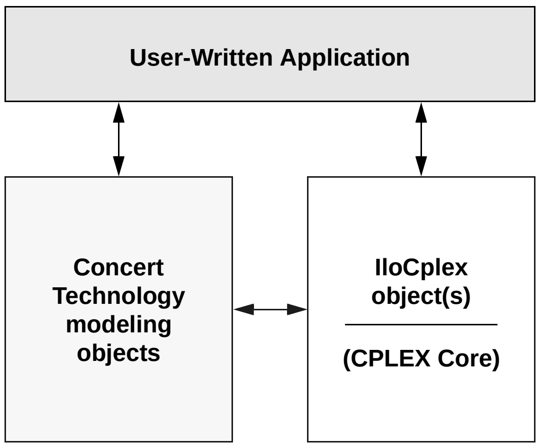

Figure 1.1 A View of CPLEX Concert Technology Library

Licenses

CPLEX runs under the control of the ILOG License Manager (ILM). Before you can run any application program that calls CPLEX, you must have established a valid license that it can read. Licensing instructions are provided to you separately when you buy or upgrade CPLEX. Contact your local ILOG support department if this information has not been communicated to you or if you find that you need help in establishing your CPLEX 7.0 license.

Compiling and Linking

Compilation and linking instructions are provided with the files that come in the standard distribution of CPLEX for your computer platform. Check the `readme` file for details.

Creating an Application with CPLEX Concert Technology Library

The remainder of this chapter is organized by the steps most applications are likely to follow.

- ◆ First the model to be solved must be created. With Concert Technology this is done independently of CPLEX. A short introduction to model creation is provided in *Modeling an Optimization Problem with Concert Technology* on page 29. A more comprehensive discussion can be found in the *ILOG Concert Technology User's Manual*.
- ◆ When the model is ready to be solved, it is handed over to CPLEX for solving. The process for doing this is shown in *Solving Concert Technology Models with IloCplex* on page 33, which includes a survey of the `IloCplex` interface for controlling the optimization. Individual controls are discussed in the chapters explaining the individual optimizers.
- ◆ *Accessing Solution Information* on page 37, shows you how to access and interpret results from the optimization after solving the model.
- ◆ After analyzing the results, you may make changes to the model and study their effect. The way to perform such changes and how CPLEX deals with them is explained in *Modifying a Model* on page 41.
- ◆ *Handling Errors* on page 43, discusses the error handling and debugging support provided by Concert Technology and CPLEX.
- ◆ In *Example: Dietary Optimization* on page 45, an example program is presented.

Not covered in this chapter are advanced functions, such as the use of callbacks for querying data about an ongoing optimization and for controlling the optimization itself. Callbacks and advanced functions are discussed in Chapter 8, *More About Using ILOG CPLEX*.

Modeling an Optimization Problem with Concert Technology

In this section we will only give a brief introduction to using Concert Technology for modeling optimization problems to be solved by `IloCplex`. For a more complete overview, see the *ILOG Concert Technology User's Manual*.

Modeling Classes

A Concert Technology model consists of a set of C++ objects. Each variable, each constraint, each SOS set, and the objective function in a model are represented by an object of the appropriate Concert Technology class. We refer to these objects as modeling objects.

Creating the Environment—The IloEnv Object

Before creating modeling objects, an object of class `IloEnv` must be constructed. We refer to this object as the environment object. It is constructed with the statement:

```
IloEnv env;
```

which is usually the first Concert Technology statement in an application. At the end, the environment must be closed by calling:

```
env.end();
```

This is usually the last Concert Technology statement in an application. The `end()` method must be called because, like most Concert Technology classes, the`IloEnv` class is a handle class. This means that `IloEnv` objects are really only pointers to implementation objects. Implementation objects are destroyed by calling the `end()` method. Failing to call the `end()` method can result in memory leaks. Please see the *ILOG Concert Technology User's Manual* and the *ILOG Concert Technology Reference Manual* for more details about handle classes in Concert Technology.

Users familiar with the callable C library are cautioned not to confuse the Concert Technology environment object with the CPLEX environment object of type `CPXENVptr`, used for setting CPLEX parameters. Such an object is not needed with Concert Technology, as parameters are handled directly by each instance of the `IloCplex` class. Thus, when talking about the environment in Concert Technology, we always refer to the object of class `IloEnv` required for all other Concert Technology objects.

Defining Variables and Expressions—The IloNumVar Object

Probably the first modeling class you will need is `IloNumVar`. Objects of this class represent modeling variables. They are described by the lower and upper bound for the variable, and a type which can be one of `ILOFLOAT`, `ILOINT`, or `ILOBOOL` for continuous, integer, or boolean variables, respectively. The following constructor creates an integer variable with bounds -1 and 10:

```
IloNumVar myIntVar(env, -1, 10, ILOINT);
```

The `IloNumVar` class provides methods that allow querying of the data needed to specify a variable. However, only bounds can be modified. Concert Technology provides a modeling object class `IloConversion` to change the type of a variable. This allows you to use the same variable with different types in different models.

Variables are usually used to build up expressions, which in turn are used to define the objective or constraints of the optimization problem. An expression can be explicitly written, as in

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

where `x` is assumed to be an array of variables (`IloNumVarArray`). Expressions can also be created piece by piece, with a loop:

```
IloExpr expr(env);
for (int i = 0; i < x.getSize(); ++i)
  expr += data[i] * x[i];
```

While Concert Technology supports very general expressions, only linear or piecewise linear expressions can be used in models to be solved with `IloCplex`. When you are done using an expression object (that is, you created a constraint with it) you need to delete it by calling its method `end()`, for example:

```
expr.end();
```

Declaring the Objective—The IloObjective Object

Objects of class `IloObjective` represent objective functions of optimization models. `IloCplex` may only handle models with at most one objective function, though the modeling API provided by Concert Technology does not impose this restriction. An objective function is specified by creating an instance of `IloObjective`. For example:

```
IloObjective obj(env,
                 1*x[1] + 2*x[2] + 3*x[3],
                 IloObjective::Minimize);
```

defines the objective to minimize the expression `1*x[1] + 2*x[2] + 3*x[3].`

Adding Constraints—The IloRange Object

Similarly, objects of class `IloRange` represent constraints for the format `lower bound <= expression <= upper bound`. Any floating point value or `+/- IloInfinity` can be used for the bounds. For example:

```
IloRange r1(env, 3.0, x[1] + x[2], 3.0);
```

defines the constraint `x[1] + x[2] == 3.0`.

Formulating a Problem—The IloModel Object

To formulate a full optimization problem, the objects that are part of it need to be selected. This is done by adding them to an instance of `IloModel`, the class used to represent optimization problems. For instance:

```
IloModel model(env);
model.add(obj);
model.add(r1);
```

defines a model consisting of the objective `obj`, constraint `r1`, and all the variables they use. Notice that variables need not be added to a model explicitly, as they are implicitly considered if any of the other modeling objects in the model use them. However, variables may be explicitly added to a model if desired.

For convenience, Concert Technology provides the functions `IloMinimize` and `IloMaximize` to define minimization and maximization objective functions. Also, operators `<=, ==`, and `<=` are overloaded to create `IloRange` constraints. This allows us to rewrite the above examples in a more compact and readable way:

```
IloModel model(env);
model.add(IloMinimize(env, 1*x[1] + 2*x[2] + 3*x[3]);
model.add(x[1] + x[2] == 3.0);
```

With this notation the C++ variables `obj` and `r1` need not be created.

The `IloModel` class is itself a class of modeling objects. Thus, one model can be added to another. A possible use of this is to capture different scenarios in different models, all of which are extensions to a core model. The core model could be represented as an `IloModel` object added to the `IloModel` objects that represent the individual scenarios.

Data Management Classes

Usually the data describing an optimization problem must be collected before or during the creation of the Concert Technology representation of the model. Though in principle modeling does not depend on how the data is generated and represented, this task may be facilitated by using the array or set classes provided by Concert Technology.

For example, objects of class `IloNumArray` can be used to store numerical data in arrays. Elements of the class `IloNumArray` can be accessed like elements of standard C++ arrays, but the class also offers a wealth of additional functions. For example, Concert Technology arrays are extensible; in other words they transparently adapt to the required size when new elements are added using the method `add()`. Conversely, elements can be removed from anywhere in the array with the method `remove()`. Concert Technology arrays also provide debugging support when compiled in debug mode by using `assert` to ensure that no element beyond the array bounds is accessed. Input and output operators (that is, `operator <<` and `operator >>`) are provided for arrays. For example, the code:

```
IloNumArray data(env, 3, 1.0, 2.0, 3.0);
cout << data << endl;
```

produces the following output:

```
[1.0, 2.0, 3.0]
```

When you are done using an array and want to reclaim its memory, call method `end()`, for example, `data.end()`. However, when ending the environment, all memory of arrays belonging to the same environment is returned to the system as well. Thus, in practice you do not call `end()` on an array (or any other Concert Technology object) just before calling `env.end()`.

The constructor for arrays specifies that an array of size 3 with elements 1.0, 2.0, and 3.0 is constructed. This output format can be read back in with, for example:

```
cin >> data;
```

The example at the end of this chapter takes advantage of this function and reads the problem data from a file.

Finally, we want to point out that Concert Technology provides the template class `IloArray<X>` to create array classes for your own type X. This can be used to generate multidimensional arrays. All the functions described above are supported for `IloArray` classes except for input/output, which depends on the input and output operator being defined for type X.

Solving Concert Technology Models with IloCplex

CPLEX generally does not need to be involved while you create your model. However, once the model is set up, it is time to create your `cplex` object, that is, an instance of the class `IloCplex`, to be used to solve the model. `IloCplex` is a class derived from `IloAlgorithm`.. There are other Concert Technology algorithm classes, also derived from `IloAlgorithm`.. Some models might also be solved by using other algorithms, such as the class `IloSolver` for constraint programming, or by using a hybrid algorithm consisting of both `IloSolver` and CPLEX. Some models, on the other hand, cannot be solved with CPLEX.

The makeup of the model determines whether or not CPLEX can be used to solve it. More precisely, in order to be handled by `IloCplex` objects, a model may only consist of modeling objects of the following classes:

Table 1.1 *Concert Technology Modeling Objects*

To model:	**Use:**
numerical variables	objects of class `IloNumVar`, as long as they are not constructed with a list of feasible values
semi-continuous variable	objects of class `IloSemiContVar`
linear objective functions	objects of class `IloObjective` with linear or piecewise linear expressions
linear constraints	objects of class `IloRange` with linear or piecewise linear expressions
variable type conversions	objects of class `IloConversion`

Table 1.1 *Concert Technology Modeling Objects (Continued)*

To model:	Use:
special ordered sets of type 1	objects of class `IloSOS1`
special ordered sets of type 2	objects of class `IloSOS2` and
constraints	objects of class `IloAnd`.

For a description of special ordered sets see *Using Special Ordered Sets (SOS)* on page 168. The last class, `IloAnd`, is listed for completeness only and is generally not used with CPLEX, except with the class `IloSolution`, as described in the *ILOG Concert Technology User's Manual*.

Extracting a Model

In this manual we describe only one optimization model and use only one instance of `IloCplex` at a time to solve the model. Consequently, we talk about these as the model and the `cplex` object. It should be noted, however, that in Concert Technology an arbitrary number of models and algorithm objects can be created, provided you have enough licenses. The `cplex` object can be created using the constructor:

```
IloCplex cplex(env);
```

To use it to solve the model, the model must first be extracted to `cplex` by calling:

```
cplex.extract(model);
```

This method copies the data from the model into the appropriate optimized data structures, which CPLEX uses for solving the problem. It does so by extracting each of the modeling objects added to the model and each of the objects referenced by them. For every extracted modeling object, corresponding data structures are created internally in the `cplex` object. For readers familiar with the sparse matrix representation used internally by CPLEX, a variable becomes a column and a constraint becomes a row. As we will discuss later, these data structures are kept synchronized with the modeling objects even if the modeling objects are modified.

If you consider a variable to be part of your model, even though it is not (initially) used in any constraint, you should add this variable explicitly to the model. This ensures that the variable will be extracted. This may also be important if you query solution information for the variable, since solution information is available only for modeling objects that are known to CPLEX because they have been extracted from a model.

If you feel uncertain about whether or not an object will be extracted, you can add it to the model to be sure. Even if an object is added multiple times, it will only be extracted once and thus will not slow the solution process down.

Since the sequence of creating the `cplex` object and extracting the model to it is such a common one, `IloCplex` provides the shortcut:

```
IloCplex cplex(model);
```

This is completely equivalent to separate calls and ensures that the environment used for the `cplex` object will be the same as that used for the model when it is extracted, as required by Concert Technology. The shortcut uses the environment from the model to construct the `cplex` object before extraction.

Solving a Model

Once the model is extracted to the `cplex` object, you are ready to solve it. This is done by calling

```
cplex.solve();
```

For most problems this is all that is needed for solving the model. Nonetheless, CPLEX offers a variety of controls that allow you to tailor the solution process for your specific needs.

Choosing an Optimizer

The most important control is the selection of the optimizer option to use for solving LPs. Solving the extracted model with CPLEX involves solving one or a series of LPs:

- Only one LP must be solved if the extracted model is an LP itself, that is, if it does not contain integer, boolean, semi-continuous or semi-integer variables, SOS, or piecewise linear functions. Chapter 4, *Solving Linear Programming Problems* discusses the algorithms available for solving LPs.
- In all other cases, the extracted problem that CPLEX sees is indeed a MIP and, in general, a series of LPs need to be solved. Method `cplex.isMIP()` returns `IloTrue` in such a case. Chapter 5, *Solving Mixed Integer Programming Problems* discusses the algorithms applied.

The optimizer option used for solving the first LP (whether or not it is the only one or just the first one in a series of problems) is controlled by calling the method:

```
cplex.setRootAlgorithm(alg);
```

where `alg` is a member of the nested enumeration type:

```
enum IloCplex::Algorithm {
  Primal,
  Dual,
  Barrier,
  NetworkPrimal,
  NetworkDual,
  DualBarrier
};
```

As a nested enumeration type, the fully qualified names that must be used in the program are `IloCplex::Primal`, `IloCplex::Dual`, and so on. Table 1.2 displays the meaning of the optimizer options defined by `IloCplex::Algorithm`.

Table 1.2 *Optimizer Options*

`IloCplex::Primal`	use the primal simplex algorithm
`IloCplex::Dual`	use the dual simplex algorithm
`IloCplex::Barrier`	use the barrier algorithm. The type of crossover performed after the barrier algorithm is determined by parameter `IloCplex::BarCrossAlg`.
`IloCplex::NetworkPrimal`	use the primal network simplex algorithm on an embedded network followed by the primal simplex algorithm on the entire problem
`IloCplex::NetworkDual`	use the primal network simplex algorithm on an embedded network followed by the dual simplex algorithm on the entire problem
`IloCplex::DualBarrier`	use the dual simplex algorithm up to the simplex iteration limit, if the LP is not solved by then switch to the barrier algorithm. This option is available only for MIPs.

If the extracted model contains more than one LP, the algorithm for solving all but the first LP is controlled by calling method `cplex.setNodeAlgorithm(alg)`. The current setting for the root and node algorithm can be queried using methods:

```
IloCplex::Algorithm IloCplex::getRootAlgorithm() const;
IloCplex::Algorithm IloCplex::getNodeAlgorithm() const;
```

Controlling CPLEX Optimizers

Though CPLEX defaults will prove sufficient to solve most of the problems, CPLEX offers a variety of parameters to control various algorithmic choices. ILOG CPLEX parameters can assume values of type `bool`, `num`, `int`, and `string`. `IloCplex` provides four categories of parameters that are listed in the nested enumeration types `IloCplex::BoolParam`, `IloCplex::IntParam, IloCplex::NumParam, IloCplex::StringParam`.

To access the *current* value of a parameter that interests you from the Concert Technology Library, use the method `getParam`. To access the default value of a parameter, use the method `getDefault`. Use the methods `getMin` and `getMax` to access the minimum and maximum values of `num` and `int` type parameters.

Some integer parameters are tied to nested enumerations that define symbolic constants for the values the parameter may assume. In particular, these enumeration types are: `IloCplex::MIPEmphasisType`, `IloCplex::VariableSelect`, `IloCplex::NodeSelect`, `IloCplex::PrimalPricing`, and

`IloCplex::DualPricing`. They are used for parameters `IloCplex::MIPEmphasis`, `IloCplex::VarSel`, `IloCplex::NodeSel`, `IloCplex::PPriInd`, and `IloCplex::DPriInd`, respectively. Only the parameter `IloCplex::MIPEmphasis` may be of importance for general use.

There are, of course, routines in the Concert Technology Library to set these parameters. Use the following methods to set the values of CPLEX parameters:

```
IloCplex::setParam(BoolParam, value);
IloCplex::setParam(IntParam, value);
IloCplex::setParam(NumParam, value);
IloCplex::setParam(StringParam, value);
```

For example, the numerical parameter `IloCplex::EpOpt` controlling the optimality tolerance for the simplex algorithms can be set to 0.0001 by calling

```
cplex.setParam(IloCplex::EpOpt, 0.0001);
```

The *ILOG CPLEX Reference Manual* documents the type of each parameter (`bool`, `int`, `num`, `string`) along with the Concert Technology enumeration value, symbolic constant, and reference number representing the parameter.

The method `setDefaults` *resets all parameters* (except the name of the log file) to their default values, including the ILOG CPLEX callback functions. This routine resets the callback functions to `NULL`.

When solving MIPs, additional controls of the solution process are provided. Priority orders and branching directions can be used to control the branching in a static way. These are discussed in *Priority* on page 162. These controls are static in the sense that they allow you to control the solution process based on data that does not change during the solution and can thus be setup before solving the model.

Dynamic control of the solution process of MIPs is provided through control callbacks. They are discussed in *Using Callbacks* on page 293. Callbacks allow you to control the solution process based on information that is generated during the solution process.

Accessing Solution Information

Accessing Solution Status

Calling `cplex.solve()` returns a boolean indicating whether or not a feasible solution (but not necessarily the optimal one) has been found. To obtain more of the information about the model that CPLEX found during the call to the `solve()` method, `cplex.getStatus()` can be called. It returns a member of the nested enumeration type:

```
enum IloAlgorithm::Status {
  Unknown,
  Feasible,
  Optimal,
  Infeasible,
  Unbounded,
  InfeasibleOrUnbounded,
  Error
};
```

Notice that the fully qualified names have the `IloAlgorithm` prefix. Table 1.3 shows what the possible return statuses mean for the extracted model.

Table 1.3 *Algorithm Status and Information About the Model*

Return Status	Extracted Model
`Feasible`	has been proven to be feasible. A feasible solution can be queried.
`Optimal`	has been solved to optimality. The optimal solution can be queried.
`Infeasible`	has been proven to be infeasible.
`Unbounded`	has been proven to be unbounded. The notion of unboundedness adopted by `IloCplex` does not include that the model has been proven to be feasible. Instead, what has been proven is that if there is a feasible solution with objective value x^*, there exists a feasible solution with objective value x^*-1 for a minimization problem, or x^*+1 for a maximization problem.
`InfeasibleOrUnbounded`	has been proven to be infeasible or unbounded.
`Unknown`	has not been able to be processed far enough to prove anything about the model. A common reason may be that a time limit was hit.
`Error`	has not been able to be processed or an error occurred during the optimization.

As can be seen, these statuses indicate information about the model that the CPLEX optimizer was able to prove during the last call to method `solve()`. In addition, the CPLEX optimizer provides information about how it terminated. For example, it may have terminated with only a feasible but not optimal solution because it hit a limit or because a user callback terminated the optimization. Such information is accessible by calling method `cplex.getCplexStatus()`, which returns a member of the nested enumeration type `IloCplex::Status`. For more information about those statuses see the *ILOG CPLEX Reference Manual*.

Querying Solution Data

If `cplex.solve()` returns `IloTrue`, a feasible solution has been found and solution values for model variables are available to be queried. For example, the solution value for the numeric variable `var1` can be accessed as follows:

```
IloNum x1 = cplex.getValue(var1);
```

However, querying solution values variable by variable may result in ugly code. Here the use of Concert Technology arrays provides a much more compact way of accessing the solution values. Assuming your variables are stored in an`IloNumVarArray var`, you can use

```
IloNumArray x(env);
cplex.getValues(x, var);
```

to access the solution values for all variables in `var` at once. Value `x[i]` contains the solution value for variable `var[i]`.

Solution data is not restricted to the solution values of variables. It also includes values of slack variables for linear constraints and the objective value. If the extracted model does not contain an objective object, `IloCplex` assumes a 0 expression objective. The objective value is returned by calling method `cplex.getObjValue()`. Slack values are accessed with the methods `getSlack()` and `getSlacks()`, which take linear constraints as a parameter.

For LPs, solution data includes information such as dual variables and reduced cost. Such information can be queried with the methods, `getDual()`, `getDuals()`, `getReducedCost()`, and `getReducedCosts()`.

Accessing Basis Information

When solving the LPs with a simplex algorithm, that is using all but the `IloCplex::Barrier` optimizer options, basis information is available as well. Basis information can be consulted using the method `IloCplex::getStatuses()` which returns basis status information for variables and constraints.

Such information is encoded by the nested enumeration type:

```
IloCplex::BasisStatus {
  Basic,
  AtLower,
  AtUpper,
  FreeOrSuperbasic
};
```

Performing Sensitivity Analysis

The availability of a basis allows you to perform sensitivity analysis for your model. Such analysis tells you by how much you can modify your model without affecting the solution you found. The modifications supported by the sensitivity analysis function include bound changes, changes of the right hand side vector and changes of the objective function. They are analyzed by methods `IloCplex::getBoundSA()`, `IloCplex::getRHSSA()`, and `IloCplex::getObjSA()`, respectively.

Analyzing Infeasible Problems

An important feature of CPLEX is that even if no feasible solution has been found, that is, if `cplex.solve()` returns `IloFalse`, some information about the problem can be queried when solving LPs. All the methods discussed so far may successfully return information about the current (infeasible) solution CPLEX maintains.

Unfortunately, there is no simple comprehensive rule about whether or not current solution information can be queried. This is because, by default, CPLEX uses a presolve procedure to simplify the model. If, for example, the model is proven to be infeasible during the presolve, no current solution is generated by the optimizer. If, in contrast, infeasibility is only proven by the optimizer, current solution information is available to be queried. The status returned by calling `cplex.getCplexStatus()` may help to determine which case you are facing, but it is probably safer and easier to include the methods for querying solution within try/ catch statements.

When an LP has been proven to be infeasible, CPLEX provides assistance for determining the cause of the infeasibility. This is done by computing what is known as an irreducibly inconsistent set (IIS), which is a description of the minimal subproblem that is still infeasible. Here minimality is defined by the property: if you remove any of the constraints (including finite bounds), the infeasibility vanishes. An IIS is computed for an infeasible model by calling method `cplex.getIIS()`.

Solution Quality

The CPLEX optimizer uses finite precision arithmetic to compute solutions. To compensate for numerical errors due to this, tolerances are used by which the computed solution is allowed to violate feasibility or optimality conditions. Thus the solution computed by the `solve()` method may in fact slightly violate the bounds specified in the model for example. You can call:

```
IloNum violation = cplex.getQuality(IloCplex::MaxPrimalInfeas);
```

to query the maximum bound violation among all variables and slacks. If you are also interested in the variable or constraint where the maximum violation occurs, call instead:

```
IloRange maxrange;
IloNumVar maxvar;
IloNum violation = cplex.getQuality(IloCplex::MaxPrimalInfeas,
                                    &maxrange,
                                    &maxvar);
```

CPLEX will copy the variable or constraint handle in which the maximum violation occurs to `maxvar` or `maxrange` and make the other handle an empty one. The maximum primal infeasibility is only one example of a wealth of quality measures. The full list is defined by the nested enumeration type `IloCplex::Quality`. All of these can be used as a parameter for the `getQuality()` methods, though some measures are not available for all optimizer option choices.

Modifying a Model

In some applications you may want to solve the modification of another model, in order, for example, to do scenario analysis or to make adaptations based on the solution of the first model. To do this, you do not have to start a new model from scratch, but instead you can take an existing model and change it to your needs. This is done by calling the modification methods of the individual modeling objects.

When an extracted model is modified, the modification is tracked in the `cplex` object. This is done through notification. Whenever a modification method is called, `cplex` objects that have extracted the model are notified about it. The `cplex` objects then track the modification in their internal data structures.

Not only does CPLEX track all modifications of the model it has extracted, but also it tries to maintain as much solution information from a previous invocation of `solve()` as is possible and reasonable.

We already encountered what is perhaps the most important modification method, that is, the method `IloModel::add()` for adding modeling objects to a model. Conversely, you may call `IloModel::remove()` to remove a modeling object from a model. Objective functions can be modified by changing their sense and by editing their expression, or by changing their expression completely. Similarly, the bounds of constraints and their expressions can be modified. For a complete list of supported modifications, see the documentation of the individual modeling objects in the *ILOG Concert Reference Manual*.

Deleting and Removing Modeling Objects

A special type of modification is that of deleting a modeling object by calling its `end()` method. Consider, for example, the deletion of a variable. What happens if the variable you delete has been used in constraints or the objective, or has been extracted to CPLEX? Concert Technology carefully removes the deleted variable from all other modeling objects and algorithms that may keep a reference to the variable in question. This applies to any modeling object to be removed. However, user-defined handles to the removed variable are not managed by Concert Technology. Instead it is up to the user to make sure that these handles are not used after the deletion of the modeling object. The only operation allowed then is the assignment operator.

Concert Technology also provides a way to remove a modeling object from all other modeling objects and algorithms exactly the same way as when deleting it, yet without deleting the modeling object. This is done by calling the method `removeFromAll()`. This may be helpful to temporarily remove a variable from your model while keeping the option to add it back later on.

It is important to understand the difference between the above and calling `model.remove(obj)` for an object `obj`. In this case, it does not necessarily mean that `obj` is removed from the problem CPLEX maintains. Whether or not this happens depends on the removed object being referenced by yet another extracted modeling object. Usually when a constraint is removed from the extracted model, the constraint is also removed from CPLEX as well, unless it was added to the model more than once.

Consider the case where a variable is removed from CPLEX after one of the delete or remove operations discussed above. If the `cplex` object contains a simplex basis, by default the status for that variable is removed from the basis as well. If the variable happens to be basic, the operation corrupts the basis. If this is not desired, CPLEX provides a delete mode that first pivots the variable out of the basis before removing it. The resulting basis is not guaranteed to be feasible or optimal, but it will still constitute a valid basis. To select this mode, call method:

```
cplex.setDeleteMode(IloCplex::FixBasis);
```

Similarly, when removing a constraint with the `FixBasis` delete mode, CPLEX will pivot the corresponding slack or artificial variable into the basis before removing it, to assure maintaining a valid basis. In either case, if no valid basis was available in the first place, no pivot operation is performed. To set the delete mode back to its default setting, call:

```
cplex.setDeleteMode(IloCplex::LeaveBasis);
```

Changing Variable Type

The type of a variable cannot be changed by calling modification methods. Instead, Concert Technology provides the modeling class `IloConversion`, the objects of which allow you to override the type of a variable in a model. This design allows you to use the same variable in different models with different types. Consider for example `model1` containing integer variable `x`. You can then create `model2`, as a copy of `model1`, that treats `x` as a continuous variable, with the following code:

```
IloModel model2(env);
model2.add(model1);
model2.add(IloConversion(env, x, ILOFLOAT));
```

A conversion object, that is, an instance of `IloConversion`, can only specify a type for a variable that is in a model. Converting the type more than once is an error, because there is no rule about which would have precedence. However, this is not a restriction, since you can remove the conversion from a model and add a new one.

Handling Errors

In Concert Technology two kinds of errors are distinguished:

1. Programming errors, such as:
   - accessing empty handle objects
   - mixing modeling objects from different environments
   - accessing Concert Technology array elements beyond an array’s size
   - passing arrays of incompatible size to functions.

   Such errors are usually an oversight of the programmer. Once they are recognized and fixed there is usually no danger of corrupting an application. In a production version, it is not necessary to handle these kinds of errors.

   In Concert Technology such error conditions are handled using assert statements. If compiled without `-DNDEBUG`, the error check is performed and the code aborts with an error message indicating which assertion failed. A production version should then be compiled with the `-DNDEBUG` compiler option, which removes all the checking. In other words, no CPU cycles are consumed for checking the assertions.

2. Runtime errors, such as memory exhaustion.

   A correct program assumes that such failures can occur and therefore must be treated, even in a production version. In Concert Technology, if such an error condition occurs, an exception is thrown.

All exceptions thrown by Concert Technology classes (including `IloCplex`) are derived from `IloException`. Exceptions thrown by algorithm classes such as `IloCplex` are derived from its child class `IloAlgorithm::Exception`. The most common exceptions thrown by CPLEX are derived from `IloCplex::Exception`, a child class of `IloAlgorithm::Exception`.

Objects of the exception class `IloCplex::Exception` correspond to the error codes generated by the C Callable Library. The error code can be queried from a caught exception by calling method:

```
IloInt IloCplex::Exception::getStatus() const;
```

The error message can be queried by calling method:

```
const char* IloException::getMessage() const;
```

which is a virtual method inherited from the base class `IloException`. If you want to access only the message for printing to a channel or output stream, it is more convenient to use the overloaded output operator (`operator<<`) provided by Concert Technology for `IloException`.

In addition to exceptions corresponding to error codes from the C Callable Library, a `cplex` object may throw exceptions pertaining only to `IloCplex`. For example, the exception `IloCplex::MultipleObjException` is thrown if a model is extracted containing more than one objective function. Such additional exception classes are derived from class `IloCplex::Exception`; objects can be recognized by a negative status code returned when calling method `getStatus()`.

In contrast to most other Concert Technology classes, exception classes are not handle classes. Thus, the correct type of an exception is lost if it is caught by value rather than by reference (that is, using `catch(IloException& e) {...}`). This is one reason that we suggest catching `IloException` objects by reference, as demonstrated in all examples. See, for example, `ilodiet.cpp`. Some derived exceptions may carry information that would be lost if caught by value. So if you output an exception caught by reference, you may get a more precise message than when outputting the same exception caught by value.

There is a second reason for catching exceptions by reference. Some exceptions contain arrays to communicate the reason for the failure to the calling function. If this information were lost by calling the exception by value, method `end()` could not be called for such arrays and their memory would be leaked (until `env.end()` is called). After catching an expression by reference, calling the exception's method `end()` will free all the memory that may be used by arrays (or expressions) of the actual exception that was thrown.

In summary, the preferred way of catching an exception is:

```
catch (IloException& e) {
  ...
  e.end();
}
```

where `IloException` may be substituted for the desired Concert Technology exception class.

Example: Dietary Optimization

The optimization problem solved in this example is to compose a diet from a set of foods, so that the nutritional requirements are satisfied and the total cost is minimized. Example `diet.cpp` illustrates:

◆ Creating a Model Row by Row

◆ Creating a Model Column by Column

◆ Creating Multi-Dimensional Arrays with IloArray

◆ Using Arrays for Input/Output

◆ Solving the Model with IloCplex

Problem Representation

The problem contains a set of foods, which are the modeling variables; a set of nutritional requirements to be satisfied, which are the constraints; and an objective of minimizing the total cost of the food. There are two ways of looking at this problem:

◆ The problem can be modeled in a *rowwise* fashion, by entering the variables first and then adding the constraints on the variables and the objective function.

◆ The problem can be modeled in a *columnwise* fashion, by constructing a series of empty constraints and then inserting the variables into the constraints and the objective function.

Concert Technology is equally suited for both kinds of modeling; in fact, you can even mix both approaches in the same program. If a new food product is created, you can create a new variable for it regardless of how the model was originally built. Similarly, if a new nutrient is discovered, you can add a new constraint for it.

Creating a Model Row by Row

You walk into the store and compile a list of foods that are offered. For each food, you store the price per unit and the amount in stock. For some foods that you particularly like, you also set a minimum amount you would like to use in your diet. Then, for each of the foods, you create a modeling variable to represent the quantity to be purchased for your diet.

Now you get a medical book and look up which nutrients are known and relevant for you. For each nutrient, you note the minimum and maximum amounts that should be found in your diet. Also, you go through the list of foods and determine how much a food item will contribute for each nutrient. This gives you one constraint per nutrient, which can naturally be represented as a range constraint:

```
nutrMin[i] <= sum_j (nutrPer[i][j] * Buy[j]) <= nutrMax[i]
```

where `i` represents the number of the nutrient under consideration, `nutrMin[i]` and `nutrMax[i]` the minimum and maximum amount of nutrient `i` and `nutrPer[i][j]` the amount of nutrient `i` in food `j`. Finally, you specify your objective function:

```
minimize sum_j (cost[j] * Buy[j])
```

This way of creating the model is shown in the function `buildModelByRow`, in example `ilodiet.cpp`.

Creating a Model Column by Column

You start with the medical book where you compile the list of nutrients that you want to ensure are properly represented in your diet. For each of the nutrients, you create an empty constraint:

```
nutrMin[i] <= ... <= nutrMax[i]
```

where `...` is left to be filled once you walk into the store. Also, you set up the objective function to minimize the cost. We refer to constraint `i` as `range[i]` and to the objective as `cost`.

Now you walk into the store and, for each food, you check the price and nutritional content. With this data you create a variable representing the amount you want to buy of the food type and install it in the objective function and constraints. That is, you create the following column:

```
cost(foodCost[j]) "+" "sum_i" (range[i](nutrPer[i][j]))
```

where the notation “`+`” and “`sum`” indicate that you “add” the new variable `j` to the objective `cost` and constraints range[`i`]. The value in parenthesis is the linear coefficient that is used for the new variable. We chose this notation for its similarity to the syntax actually used in Concert Technology, as demonstrated in the function `buildModelByColumn`, in example `ilodiet.cpp`.

Creating Multi-Dimensional Arrays with IloArray

All data defining the problem are read from a file. The nutrients per food are stored in a two-dimensional array. Concert Technology does not provide a predefined array class; however, by using the template class `IloArray`, you can create your own two-dimensional array class. This class is defined with the type definition:

```
typedef IloArray<IloNumArray> IloNumArray2;
```

and is then ready to use, just like any predefined Concert Technology class, for example `IloNumArray`, the one-dimensional array class for numerical data.

Program Description

The main program starts by declaring the environment and terminates by calling method `end()` for the environment. The code in between is encapsulated in a try block that catches all Concert Technology exceptions and prints them to the C++ error stream `cerr`. All other exceptions are caught as well, and a simple error message is issued. The first action of the program is to evaluate command line parameters and call function usage in cases of misuse.

> ***Note:** In such cases, an exception is thrown. This ensures that* `env.end()` *is called before the program is terminated.*

Using Arrays for Input/Output

If all goes well, the input file is opened in the file `ifstream`. After that, the arrays for storing the problem data are created by declaring the appropriate variables. Then the arrays are filled by using the input operator with the data file. The data is checked for consistency and, if it fails, the program is aborted, again by throwing an exception.

After the problem data has been read and verified, we are ready to build the model. To do so we construct the model object with the declaration

```
IloModel mod(env);
```

The array `Buy` is created to store the modeling variable. Since the environment is not passed to the constructor of `Buy`, an empty handle is constructed. So at this point the variable `Buy` cannot be used.

Depending on the command line function, either `buildMethodByRow` or `buildMethodByColumn` is called. Both create the dietary model from the input data and return an array of modeling variables as an instance of the class `IloNumVarArray`. At that point, `Buy` is assigned to an initialized handle containing all the modeling variables and can be used afterwards.

Building the Model by Row

The function `buildModelByRow` implements the rowwise creation of the model. It first gets the environment from the model object passed to it. Then the modeling variables `Buy` are created. Instead of calling the constructor for the variables individually for each variable, we create the full array of variables, with the array of lower and upper bounds and the variable type as parameter. In this array, variable `Buy[i]` is created such that it has lower bound `foodMin[i]`, upper bound `foodMax[i]`, and type `type`.

The statement:

```
mod.add(IloMinimize(env, IloScalProd(Buy, foodCost)));
```

creates the objective function and adds it to the model. The `IloScalProd` function creates the expression `sum_j (Buy[j] * foodCost[j])` which is then passed to the function

`IloMinimize`. That function creates and returns the actual `IloObjective` object, which is added to the model with the call `mod.add()`.

The following loop creates the constraints of the problem one by one and adds them to the model. First the expression `sum_j (Buy[j] * nutrPer[i][j])` is created by building a Concert Technology expression. An expression variable `expr` of type `IloExpr` is created, and linear terms are added to it by using `operator+=` in a loop. The expression is used with the overloaded `operator<=` to construct a range constraint (an `IloRange` object) which is added to the model:

```
mod.add(nutrMin[i] <= expr <= nutrMax[i]);
```

After an expression has been used for creating a constraint, it is deleted by calling `expr.end()`.

Finally, the array of modeling variables `Buy` is returned.

Building the Model by Column

The function `buildModelByColumn()`implements the columnwise creation of the model. It begins by creating the array of modeling variables `Buy` of size 0. This is later populated when the columns of the problem are created and eventually returned.

The statement:

```
IloObjective cost = IloAdd(mod, IloMinimize(env));
```

creates a minimization objective function object with 0 expressions and adds it to the model. The objective object is created with the function `IloMinimize`. The template function `IloAdd` is used to add the objective object to the model and to return an objective object with the same type, so that we can store the objective in variable `cost`. The method `IloModel::add()` returns the modeling object as an `IloExtractable`, which cannot be assigned to a variable of a derived class such as `IloObjective`. Similarly an array of range constraints with 0 expressions is created, added to the model, and stored in array `range`.

In the following loop, the columns of the model are created one by one. First a representation of each new column is created, using the numeric column variable `col` (an instance of `IloNumColumn`), and initialized with the objective coefficient for the new variable. This coefficient is returned by `cost(foodCost[j])` which calls the overloaded `operator()` for `IloObjective` objects. Then the coefficients for the constraints are added to the column using `operator+=`. The coefficient for row `i` is created with `range[i](nutrPer[i][j])`, which calls the overloaded `operator()` for `IloRange` objects.

When a column is completely constructed, a new variable is created for it and added to the array of modeling variables `Buy`. The construction of the variable is performed by the constructor:

```
IloNumVar(col, foodMin[j], foodMax[j], type)
```

which creates the new variable with lower bound `foodMin[j]`, upper bound `foodMax[j]` and type `type`, and adds it to the existing objective and ranges with the coefficients specified in column `col`. Again, after creating the variable for this column, the `IloColumn` object is deleted by calling `col.end()`.

Solving the Model with IloCplex

After the model has been populated, we are ready to create the `cplex` object and extract the model to it by calling:

```
IloCplex cplex(mod);
```

It is then ready to solve the model, but for demonstration purposes we first write the extracted model to file `diet.lp`. Doing so can help you debug your model, as the file contains exactly what CPLEX sees. If it does not match what you expected, it will probably help you locate the code that generated the wrong part.

The model is then solved by calling method `solve()`. Finally, the solution status and solution vector are output to the output channel `cplex.out()`. By default this channel is initialized to `cout`. All logging during optimization is also output to this channel. To turn off logging, you would set the `out()` stream of `cplex` to a null stream by calling `cplex.setOut(env.getNullStream())`.

Complete Program

The complete program, `ilodiet.cpp`, shown here is also provided online, in the standard distribution.

Notes:

- *All the definitions needed for a CPLEX Concert Technology application are imported by including the file* `<ilcplex/ilocplex.h>`.
- *The line* `ILOSTLBEGIN` *is a macro that is needed for portability. Microsoft Visual C++ code varies, depending on whether you use the STL or not. This macro allows you to switch between both types of code without the need to otherwise change your source code.*
- *Function usage is called in case the program is executed with incorrect command line arguments.*

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

void usage(const char* name) {
  cerr << endl;
  cerr << "usage:   " << name << " [options] <file>" << endl;
  cerr << "options: -c  build model by column" << endl;
  cerr << "         -i  use integer variables" << endl;
  cerr << endl;
}

typedef IloArray<IloNumArray> IloNumArray2;

IloNumVarArray
buildModelByRow(IloModel mod,
                const IloNumArray foodMin,
                const IloNumArray foodMax,
                const IloNumArray foodCost,
                const IloNumArray nutrMin,
                const IloNumArray nutrMax,
                const IloNumArray2& nutrPer,
                IloNumVar::Type type) {
  IloEnv env = mod.getEnv();

  IloNumVarArray Buy (env, foodMin, foodMax, type);

  IloInt i, j;
  IloInt n = foodCost.getSize();
  IloInt m = nutrMin.getSize();

  mod.add(IloMinimize(env, IloScalProd(Buy,foodCost)));
  for (i = 0; i < m; i++) {
     IloExpr expr(env);
     for (j = 0; j < n; j++)  expr += Buy[j] * nutrPer[i][j];
     mod.add(nutrMin[i] <= expr <= nutrMax[i]);
  }

  return (Buy);
}

IloNumVarArray
buildModelByColumn(IloModel mod,
                   const IloNumArray foodMin,
                   const IloNumArray foodMax,
                   const IloNumArray foodCost,
                   const IloNumArray nutrMin,
                   const IloNumArray nutrMax,
                   const IloNumArray2& nutrPer,
                   IloNumVar::Type type) {
  IloEnv env = mod.getEnv();

  IloNumVarArray Buy(env);

  IloInt i, j;
  IloInt n = foodCost.getSize();
  IloInt m = nutrMin.getSize();

  IloObjective  cost  = IloAdd(mod, IloMinimize(env));
  IloRangeArray range = IloAdd(mod, IloRangeArray(env, nutrMin, nutrMax));

  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));
  }

  return (Buy);
}

int
main(int argc, char **argv)
{
  IloEnv env;

  try {
    const char*     filename  = "../../../examples/data/diet.dat";
    IloBool         byColumn  = IloFalse;
    IloNumVar::Type varType   = ILOFLOAT;

    IloInt i;

    for (i = 1; i < argc; ++i) {
      if (argv[i][0] == ’-’) {
        switch (argv[i][1]) {
        case ’c’:
          byColumn = IloTrue;
          break;
        case ’i’:
          varType = ILOINT;
          break;
        default:
          usage(argv[0]);
          throw (-1);
        }
      }
      else {
        filename = argv[i];
        break;
      }
    }

    ifstream file(filename);
    if ( !file ) {
      cerr << "ERROR: could not open file ’" << filename << "’ for reading" <<
endl;
      usage(argv[0]);
      throw (-1);
    }

    // model data

    IloNumArray  foodCost(env), foodMin(env), foodMax(env);
    IloNumArray  nutrMin(env), nutrMax(env);
    IloNumArray2 nutrPer(env);

    file >> foodCost >> foodMin >> foodMax;
    file >> nutrMin >> nutrMax;
    file >> nutrPer;

    IloInt nFoods = foodCost.getSize();
    IloInt nNutr  = nutrMin.getSize();

    if ( foodMin.getSize() != nFoods ||
         foodMax.getSize() != nFoods ||
         nutrPer.getSize() != nNutr  ||
         nutrMax.getSize() != nNutr    ) {
      cerr << "ERROR: Data file '" << filename
           << "' contains inconsistent data" << endl;
      throw (-1);
    }

    for (i = 0; i < nNutr; ++i) {
      if (nutrPer[i].getSize() != nFoods) {
        cerr << "ERROR: Data file '" << argv[0]
             << "' contains inconsistent data" << endl;
        throw (-1);
      }
    }

    // Build model

    IloModel       mod(env);
    IloNumVarArray Buy;
    if ( byColumn ) {
      Buy = buildModelByColumn(mod, foodMin, foodMax, foodCost,
                               nutrMin, nutrMax, nutrPer, varType);
    }
    else {
      Buy = buildModelByRow(mod, foodMin, foodMax, foodCost,
                            nutrMin, nutrMax, nutrPer, varType);
    }

    // Solve model

    IloCplex cplex(mod);
    cplex.exportModel("diet.lp");

    cplex.solve();
    cplex.out() << "solution status = " << cplex.getStatus() << endl;

    cplex.out() << endl;
    cplex.out() << "cost   = " << cplex.getObjValue() << endl;
    for (i = 0; i < foodCost.getSize(); i++)
      cplex.out() << "  Buy" << i << " = " << cplex.getValue(Buy[i]) << endl;
  }
  catch (IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  catch (...) {
    cerr << "Error" << endl;
  }
  env.end();

  return 0;
}
```

C H A P T E R

2

Using the ILOG CPLEX Callable Library

This chapter describes how to write C programs using the ILOG ILOG CPLEX Callable Library. It includes sections on:

◆ Architecture of the CPLEX Callable Library, including information on licensing and on compiling and linking your programs

◆ Using the Callable Library in an Application

◆ ILOG CPLEX Programming Practices

◆ Managing Parameters from the Callable Library

◆ Example: Dietary Optimization

Architecture of the CPLEX Callable Library

ILOG CPLEX includes a callable C library that makes it easier to develop applications to optimize, to modify, and to interpret the results of mathematical programming problems whether linear, mixed integer, or convex quadratic ones.

You can use the Callable Library to write applications that conform to many modern computer programming paradigms, such as client-server applications within distributed computing environments, multithreaded applications running on multiple processors, applications linked to database managers, or applications using flexible graphic user interface builders, just to name a few.

The Callable Library together with the ILOG CPLEX database make up the ILOG CPLEX *core*, as you see in Figure 2.1. The ILOG CPLEX database includes the computing environment, its communication channels, and your problem objects. You will associate the core with your application by calling library routines.

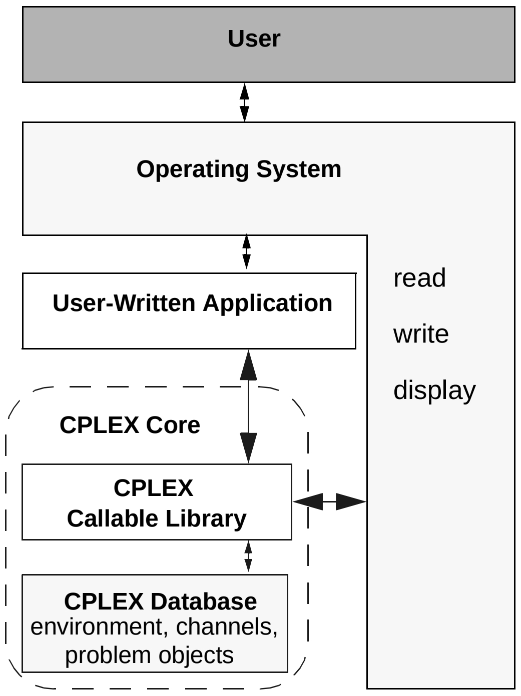

Figure 2.1 *A view of the ILOG CPLEX world*

The ILOG CPLEX Callable Library itself contains routines organized into several categories:

- *optimization routines* enable you to define a problem, optimize it, and generate results;
- *utility routines* handle application programming issues;
- *problem modification routines* let you change a problem after you have created it within the ILOG CPLEX database;
- *problem query routines* access information about a problem after you have created it;

- *file reading and writing routines* move information from the file system of your operating system into your application, or from your application into the file system;
- *parameter routines* enable you to query, set, or modify parameter values maintained by ILOG CPLEX.

Licenses

CPLEX runs under the control of the ILOG License Manager (ILM). Before you can run any application program that calls CPLEX, you must have established a valid license that it can read. Licensing instructions are provided to you separately when you buy or upgrade CPLEX. Contact your local ILOG support department if this information has not been communicated to you or if you find that you need help in establishing your CPLEX 7.0 license.

Compiling and Linking

Compilation and linking instructions are provided with the files that come in the standard distribution of CPLEX for your computer platform. Check the `readme` file for details.

Using the Callable Library in an Application

This section tells you how to use the Callable Library in your own applications. Briefly, you must initialize the ILOG CPLEX environment, instantiate a problem object, and fill it with data. Then your application calls one of the ILOG CPLEX optimizers to optimize your problem. Optionally, your application can also modify the problem object and re-optimize it. ILOG CPLEX is designed to support this sequence of operations—modification and re-optimization of linear programming problems (LPs)—efficiently by reusing the current *basis* of a problem as its starting point (when applicable). After it finishes using ILOG CPLEX, your application must free the problem object and release the ILOG CPLEX environment it has been using. The following sections explain these steps in greater detail.

Initialize the ILOG CPLEX Environment

ILOG CPLEX needs certain internal data structures to operate. In your own application, you use a routine from the Callable Library to initialize these data structures. You must initialize these data structures *before* your application calls any other routine in the ILOG CPLEX Callable Library.

To initialize a ILOG CPLEX environment, you must use the routine `CPXopenCPLEX()`.

This routine checks for a valid ILOG CPLEX license and then returns a C pointer to the ILOG CPLEX environment that is creates. Your application then passes this C pointer to other ILOG CPLEX routines (except `CPXmsg()`). As a developer, you decide for yourself whether the variable containing this pointer should be global or local in your application

A multithreaded application needs multiple ILOG CPLEX environments. Consequently, ILOG CPLEX allows more than one environment to exist at a time; each one consumes a licensed process.

Instantiate the Problem Object

Once you have initialized a ILOG CPLEX environment, your next step is to *instantiate* (that is, create and initialize) a *problem object* by calling `CPXcreateprob()`. This routine returns a C pointer to the problem object. Your application then passes this pointer to other routines of the Callable Library.

Most applications will use only one problem object, though ILOG CPLEX allows you to create multiple problem objects within a given ILOG CPLEX environment. Similarly, most applications create only one ILOG CPLEX environment, although ILOG CPLEX allows *l* environments, where *l* is the number of licensed ILOG CPLEX users on your system.

Put Data in the Problem Object

When you instantiate a problem object, it is originally empty. In other words, it has no constraints, no variables, and no coefficient matrix. ILOG CPLEX offers you several alternative ways to put data into an empty problem object (that is, to *populate* your problem object).

- ◆ You can assemble arrays of data and then call `CPXcopylp()` to copy the data into the problem object.
- ◆ You can make a sequence of calls, in any convenient order, to these routines:
  - `CPXnewcols();`
  - `CPXnewrows();`
  - `CPXaddcols();`
  - `CPXaddrows();`
  - `CPXchgcoeflist();`
- ◆ If data already exist in MPS or LP format in a file, you can call `CPXreadcopyprob()` to read that file and copy the data into the problem object. (MPS—Mathematical Programming System—is an industry-standard format for organizing data in mathematical programming problems. LP—linear programming—is a ILOG CPLEX-specific format for expressing linear programming problems as equations or inequalities. *Understanding File Formats* on page 264 explains these formats in greater detail.)

Optimize the Problem

Call one of the ILOG CPLEX optimizers to solve the problem object that you have instantiated and populated. *Choosing an Optimizer for Your LP Problem* on page 96 explains in greater detail how to choose an appropriate optimizer for your problem.

Change the Problem Object

In analyzing a given mathematical program, you may make changes in a model and study their effect. As you make such changes, you must keep ILOG CPLEX informed about the modifications so that ILOG CPLEX can efficiently re-optimize your changed problem. Always use the *problem modification routines* from the Callable Library to make such changes and thus keep ILOG CPLEX informed. In other words, do *not* change a problem by altering the original data arrays and calling `CPXcopylp()` again. That tempting strategy usually will not make the best use of ILOG CPLEX. Instead, modify your problem by means of the problem modification routines.

For example, let's say a user has already solved a given problem and then changes the upper bound on a variable by means of an appropriate call to the Callable Library. ILOG CPLEX will then begin any further optimization from the previous optimal *basis*. If that basis is still optimal with respect to the new bound, then ILOG CPLEX will return that information without even needing to refactor the basis.

Destroy the Problem Object

Use the routine `CPXfreeprob()` to destroy a problem object when your application no longer needs it.

Release the ILOG CPLEX Environment

After all the calls from your application to the ILOG CPLEX Callable Library are complete, you must release the ILOG CPLEX environment by calling the routine `CPXcloseCPLEX()`. This routine tells ILOG CPLEX that:

- ◆ all application calls to the Callable Library are complete;
- ◆ ILOG CPLEX should release any memory allocated by ILOG CPLEX for this environment;
- ◆ the application has relinquished the ILOG CPLEX license for this run, thus making the license available to the next user.

ILOG CPLEX Programming Practices

This section lists the programming practices we observe in developing and maintaining the ILOG CPLEX Callable Library.

The ILOG CPLEX Callable Library supports modern programming practices. It uses no external variables. Indeed, no global nor static variables are used in the library so that the Callable Library is fully reentrant and thread-safe. The names of all library routines begin with the three-character prefix `CPX` to prevent namespace conflicts with your own routines or with other libraries. Also to avoid clutter in the namespace, there is a minimal number of routines for setting and querying parameters.

Variable Names and Calling Conventions

Routines in the ILOG CPLEX Callable Library obey the C programming convention of *call by value* (as opposed to call by reference, for example, in FORTRAN and BASIC). If a routine in the Callable Library needs the address of a variable in order to change the value of the variable, then that fact is documented in the *ILOG CPLEX Reference Manual* by the suffix `_p` in the variable name in the synopsis of the routine. In C, you create such values by means of the `&` operator to take the address of a variable and to pass this address to the Callable Library routine.

For example, let's look at the synopses for two routines, `CPXgetobjval()` and `CPXgetx()`, as they are documented in the *ILOG CPLEX Reference Manual* to clarify this calling convention. Here is the synopsis of the routine `CPXgetobjval()`:

```
int CPXgetobjval (CPXENVptr env, CPXLPptr lp, double *objval_p)
```

In that routine, the third parameter is a pointer to a variable of type `double`. To call this routine from C, declare:

```
double objval;
```

Then call `CPXgetobjval()` in this way:

```
status = CPXgetobjval (env, lp, &objval);
```

In contrast, here is the synopsis of the routine `CPXgetx()`:

```
int CPXgetx (CPXENV env, CPXLPptr lp, double *x, int begin, int end)
```

You call it by creating a double-precision array by means of either one of two methods. The first method dynamically allocates the array, like this:

```
double *x = NULL;
x = (double *) malloc (100*sizeof(double));
```

The second method declares the array as a local variable, like this:

```
double x[100];
```

Then to see the optimal values for columns 5 through 104, for example, you could write this:

```
status = CPXgetx (env, lp, x, 5, 104);
```

The variable `objval_p` in the synopsis of `CPXgetobjval()` and the variable `x` in the synopsis of `CPXgetx()` are both of type `(double *)`. However, the suffix `_p` in the parameter `objval_p` indicates that you should use an address of a single variable in one call, while the lack of `_p` in `x` indicates that you should pass an array in the other.

For guidance about how to pass values to ILOG CPLEX routines from application languages such as FORTRAN or BASIC that conventionally call by reference, see *Call by Reference* on page 69 in this manual, and consult the documentation for those languages.

Data Types

In the Callable Library, ILOG CPLEX defines a few special data types for specific ILOG CPLEX objects, as you see in Table 2.1.

Table 2.1 *Special data types in the ILOG CPLEX Callable Library*

Data type	Is a pointer to	Declaration	Set by calling
`CPXENVptr`	ILOG CPLEX environment	`CPXENVptr env;`	`CPXopenCPLEX()`
`CPXLPptr`	problem object	`CPXLPptr lp;`	`CPXcreateprob()`
`CPXNETptr`	problem object	`CPXNETptr net;`	`CPXNETcreateprob()`
`CPXCHANNELptr`	message channel	`CPXCHANNELptr channel;`	`CPXgetchannels()` `CPXaddchannel()`

When any of these special variables are set to a value returned by an appropriate routine, that value can be passed directly to other ILOG CPLEX routines that require such parameters. The actual internal type of these variables is a memory address (that is, a pointer); this address uniquely identifies the corresponding object. If you are programming in a language other than C, you should choose an appropriate integer type or pointer type to hold the values of these variables.

Ownership of Problem Data

The ILOG CPLEX Callable Library does not take ownership of user memory. All arguments are copied from your user-defined arrays into ILOG CPLEX-allocated memory. ILOG CPLEX manages all problem-related memory. After you call a ILOG CPLEX routine that copies data into a ILOG CPLEX problem object, you can free the memory you used as arguments to the copying routine.

Copying in MIP and QP

If you are licensed to use the ILOG CPLEX Mixed Integer Optimizer, the routine `CPXcopyctype()` copies information about variable types in a mixed integer programming application (MIP).

If you are licensed to use the ILOG CPLEX Barrier Optimizer, the routines `CPXcopyqsep()` and `CPXcopyquad()` are for copying information about quadratic objective coefficients in a convex quadratic programming application (QP).

Problem Size and Memory Allocation Issues

As we indicated in *Change the Problem Object* on page 59, after you have created a problem object by calling `CPXcreateprob()`, you can modify the problem in various ways through calls to routines from the Callable Library. There is no need for you to allocate extra space in anticipation of future problem modifications. Any limit on problem size is determined by system resources and the underlying implementation of the system function `malloc()`—not by artificial limits in ILOG CPLEX.

As you modify a problem object through calls to modification routines from the Callable Library, ILOG CPLEX automatically handles memory allocations to accommodate the increasing size of the problem. In other words, you do not have to keep track of the problem size nor make corresponding memory allocations yourself as long as you are using library modification routines such as `CPXaddrows() or CPXaddcols()`.

However, the sequence of Callable Library routines that you invoke can influence the efficiency of memory management. Likewise, parameters controlling row growth `(CPX_PARAM_ROWGROWTH)`, column growth `(CPX_PARAM_COLGROWTH)`, and nonzero growth (`CPX_PARAM_NZGROWTH`) can also influence how efficiently ILOG CPLEX allocates memory to accommodate the problem object. These growth parameters determine how much extra space ILOG CPLEX allocates in its internal structures when additions to a problem object increase the size of the problem object so that it exceeds currently allocated space.

Table 2.2 *Default values of ILOG CPLEX growth parameters*

Parameter	Default value
`CPX_PARAM_ROWGROWTH`	100
`CPX_PARAM_COLGROWTH`	100
`CPX_PARAM_NZGROWTH`	500

Table 2.2 shows you the default values of these growth parameters. At these default values, if an application populates the problem object one row at a time, then CPLEX will cache the row additions until an updated problem is needed, for example when a query or optimization function is called. Similarly, it will cache column-based additions after 100 columns, and nonzero-based arrays when the additions of coefficients produces another 500 nonzeros to add to the matrix. *Memory Management and Problem Growth* on page 103 offers guidelines about performance tuning with these parameters.

Status and Return Values

The Callable Library routine `CPXopenCPLEX()` returns a pointer to a ILOG CPLEX environment. In case of failure, it returns a `NULL` pointer. The parameter `*status_p` (that is, one of its arguments) is set to 0 if the routine is successful; in case of failure, that parameter is set to a nonzero value that indicates the reason for the failure. Each failure value is unique and documented in the *ILOG CPLEX Reference Manual.*

The Callable Library routine `CPXcreateprob()` returns a pointer to a ILOG CPLEX problem object and sets its parameter `*status_p to 0 (zero)` to indicate success. In case of failure, it returns a `NULL` pointer and sets `*status_p` to a nonzero value indicating the reason for the failure.

Some query routines in the Callable Library return a nonzero value when they are successful. For example, `CPXgetnumcols()` returns the number of columns in the constraint matrix (that is, the number of variables in the problem object). However, most query routines return `0` (zero) indicating success of the query and entail one or more parameters (such as a buffer or character string) to contain the results of the query. For example, `CPXgetrowname()` returns the name of a row in its `name` parameter.

Most other routines in the Callable Library return an integer value, `0` (zero) indicating success of the call. A nonzero return value indicates a failure. Each failure value is unique and documented in the *ILOG CPLEX Reference Manual.*

We strongly recommend that your application check the status—whether the status is indicated by the return value or by a parameter—of the routine that it calls before it proceeds.

Symbolic Constants

Most ILOG CPLEX routines return or require values that are defined as symbolic constants in the header file (that is, the include file) `cplex.h.` We highly recommend this practice of using symbolic constants, rather than hard-coded numeric values. Symbolic names improve the readability of calling applications. Moreover, if numeric values happen to change in subsequent releases of the product, the symbolic names will remain the same, thus making applications easier to maintain.

Parameter Routines

You can set many parameters in the ILOG CPLEX environment to control ILOG CPLEX operation. The values of these parameters may be integer, double, or character strings, so there are sets of routines for accessing and setting them. Table 2.3 shows you the names and purpose of these routines. Each of these routines accepts the same first argument: a pointer to the ILOG CPLEX environment (that is, the pointer returned by `CPXopenCPLEX()`). The second argument of each of those parameter routines is the parameter number, a symbolic constant defined in the header file, `cplex.h`. *Managing Parameters from the Callable Library* on page 70 offers more details about parameter settings.

Table 2.3 *Callable Library routines for parameters in the ILOG CPLEX environment*

Type	Change value	Access current value	Access default, max, min
integer	`CPXsetintparam()`	`CPXgetintparam()`	`CPXinfointparam()`
double	`CPXsetdblparam()`	`CPXgetdblparam()`	`CPXinfodblparam()`
string	`CPXsetstrparam()`	`CPXgetstrparam()`	`CPXinfostrparam()`

Null Arguments

Certain ILOG CPLEX routines that accept optional arguments allow you to pass a `NULL` pointer in place of the optional argument. The documentation of those routines in the *ILOG CPLEX Reference Manual* indicates explicitly whether `NULL` pointer arguments are acceptable. (Passing null arguments is an effective way to avoid allocating unnecessary arrays.)

Row and Column References

Consistent with standard C programming practices, in ILOG CPLEX an array containing `k` items will contain these items in locations `0` (zero) through `k-1`. Thus a linear program with `m` rows and `n` columns will have its rows indexed from `0` to `m-1`, and its columns from `0` to `n-1`.

Within the linear programming data structure, the rows and columns that represent constraints and variables are referenced by an *index number.* Each row and column may optionally have an associated name. If you add or delete rows, the index numbers usually change. However, ILOG CPLEX updates the names so that each row or column index will correspond to the correct row or column name. Double checking names against index numbers is the only sure way to determine which changes may have been made to matrix indices in such a context. The routines `CPXgetrowindex()` and `CPXgetcolindex()` translate names to indices.

If additions or deletions to the constraint matrix are few, then:

- for deletions, ILOG CPLEX decrements each reference index above the deletion point, and
- for additions, ILOG CPLEX makes all additions at the end of the existing range.

Here is an example to illustrate how rows are renumbered when rows `k+1` to `l-1` are deleted. That is, `l-(k+1)` elements are deleted in the example:

Rows before deletion	`0, . . . k,`	`k+1,`	`. . . ,`	`l-1,`	`l,`	`. . .,`	`end`
Rows after deletion	`0, . . . k,`	`k+1,`	`. . . ,`	`[end-(l-(k+1))]`			

Character Strings

You can pass character strings as parameters to various ILOG CPLEX routines, for example, as row or column names. The Interactive Optimizer truncates output strings usually at 18 characters. Routines from the Callable Library truncate strings at 255 characters in output *text* files (such as MPS, LP, and SOS text files) but not in *binary* SAV files. Routines from the Callable Library also truncate strings at 255 characters in names that occur in messages. Routines of the Callable Library that produce log files, such as the simplex iteration log file or the MIP node log file, truncate at 16 characters. The Callable Library routine `CPXwritesol()` truncates character strings in binary solution files at 8 characters and in text solution files at 16 characters. Input, such as names read from LP and MPS files or typed interactively by the `enter` command, are truncated to 255 characters. However, we do not recommend that you rely on this truncation because unexpected behavior may result.

Checking Problem Data

If you inadvertently make an error entering problem data, the problem object will not correspond to your intentions. One possible result may be a segmentation fault or other disruption of your application. In other cases, ILOG CPLEX may solve a different model from the one you intended, and that situation may or may not result in error messages from ILOG CPLEX.

Using the Data Checking Parameter

To help you detect this kind of error, you can set the parameter `CPX_PARAM_DATACHECK` to the value `CPX_ON` to activate additional checking of array arguments for `CPXcopy...()`, `CPXread...()`, and `CPXchg...()` functions. The additional checks include:

- invalid sense/ctype/sostype values
- indexes out of range, for example, `rowind >= numrows`

- duplicate entries
- `matbeg` or `sosbeg` array with decreasing values
- NANs in double arrays
- NULLs in name arrays

When the parameter is set to `CPX_OFF`, only simple checks, for example checking for the existence of the environment, are performed.

Using Diagnostic Routines for Debugging

ILOG CPLEX also provides diagnostic routines to look for common errors in the definition of problem data. In the standard distribution of ILOG CPLEX, the file `check.c` contains the source code for these routines:

- `CPXcheckcopylp()`
- `CPXcheckcopylpwnames()`
- `CPXcheckcopyqpsep()`
- `CPXcheckcopyquad()`
- `CPXcheckaddrows()`
- `CPXcheckaddcols()`
- `CPXcheckchgcoeflist()`
- `CPXcheckvals()`
- `CPXcheckcopyctype()`
- `CPXcheckcopysos()`
- `CPXNETcheckcopynet()`

Each of those routines performs a series of diagnostic tests of the problem data and issues warnings or error messages whenever it detects a potential error. To use them, you must compile and link the file `check.c`. After compiling and linking that file, you will be able to step through the source code of these routines with a debugger to help isolate problems.

If you have observed anomalies in your application, you can exploit this diagnostic capability by calling the appropriate routines just before a copying routine. The diagnostic routine may then detect errors in the problem data that could subsequently cause inexplicable behavior.

Those checking routines send all messages to one of the standard ILOG CPLEX message channels. You capture that output by setting the parameter `CPX_PARAM_SCRIND` (if you want messages directed to your screen) or by calling the routine `CPXsetlogfile()`.

Callbacks

The Callable Library supports callbacks so that you can define functions that will be called at crucial points in your application:

- ◆ during the presolve process;
- ◆ once per iteration in a linear programming routine;
- ◆ once before a node is processed in a mixed integer optimization (if the end-user is licensed for the ILOG CPLEX Mixed Integer Optimizer).

In addition, callback functions can call `CPXgetcallbackinfo()` to retrieve information about the progress of an optimization algorithm. They can also return a value to indicate whether an optimization should be aborted. `CPXgetcallbackinfo()` is the only routine of the Callable Library that a user-defined callback may call. (Of course, calls to routines not in the Callable Library are permitted.)

Using Callbacks on page 293 describes callback facilities in greater detail.

Portability

ILOG CPLEX contains a number of features to help you create Callable Library applications that can be easily ported between UNIX and Windows 95 and NT (that is, Win32) platforms.

CPXPUBLIC

All ILOG CPLEX Callable Library routines except `CPXmsg()` have the word `CPXPUBLIC` as part of their prototype. On UNIX platforms, this has no effect. On Win32 platforms, the `CPXPUBLIC` designation tells the compiler that all of the ILOG CPLEX functions are compiled with the Microsoft `__stdcall` calling convention. The exception `CPXmsg()` cannot be called by `__stdcall` because it takes a variable number of arguments. Consequently, `CPXmsg()` is declared as `CPXPUBVARARGS`; that calling convention is defined as `__cdecl` for Win32 systems.

Function Pointers

All ILOG CPLEX Callable Library routines that require pointers to functions expect the passed-in pointers to be declared as `CPXPUBLIC`. Consequently, when your application uses the ILOG CPLEX Callable Library routines `CPXaddfuncdest()`, `CPXsetlpcallbackfunc()`, and `CPXsetmipcallbackfunc()`, it must declare the user-written callback functions with the `CPXPUBLIC` designation. For UNIX systems, this has no effect. For Win32 systems, this will cause the callback functions to be declared with the `__stdcall` calling convention. For examples of function pointers and callbacks, see *Example: Using Callbacks* on page 303 and *Example: Using the Message Handler* on page 272.

CPXCHARptr and CPXVOIDptr

The types `CPXCHARptr` and `CPXVOIDptr` are used in the header file `cplex.h` to avoid the complicated syntax of using the `CPXPUBLIC` designation on functions that return `char` or `void` pointers.

File Pointers

File pointer arguments for Callable Library routines should be declared with the type `CPXFILEptr`. On UNIX platforms, this practice is equivalent to using the file pointer type. On Win32 platforms, the file pointers declared this way will correspond to the environment of the ILOG CPLEX DLL. Any file pointer passed to a Callable Library routine should be obtained with a call to `CPXfopen()` and closed with `CPXfclose()`. Callable Library routines with file pointer arguments include `CPXsetlogfile()`, `CPXaddfpdest()`, `CPXdelfpdest()`, and `CPXfputs()`. *Handling Message Channels: Callable Library Routines* on page 271 discusses most of those routines.

String Functions

Several routines in the ILOG CPLEX Callable Library make it easier to work with strings. These functions are helpful when you are writing applications in a language, such as Visual Basic, that does not allow you to dereference a pointer. The string routines in the ILOG CPLEX Callable Library are `CPXmemcpy()`, `CPXstrlen()`, `CPXstrcpy()`, and `CPXmsgstr()`.

ILOG CPLEX-Allocated Memory

The Callable Library read routines `CPXreadcopyprob()`, etc., return pointers to memory allocated by the Callable Library. The Callable Library routine `CPXfree()` frees these pointers. On UNIX systems, it is acceptable, but *not* recommended for portability reasons, to use the system call `free()`.

For more complete access to ILOG CPLEX memory management, you also have the routines `CPXmalloc()` and `CPXrealloc()`. However, you are not required to use these two routines in order to have correctly functioning Callable Library applications.

FORTRAN Interface

The Callable Library can be interfaced with FORTRAN applications. You can download examples of a FORTRAN application from the ILOG web site at `http://www.ilog.com/products/cplex`. Choose Tech Support; then choose Callable Library Examples. The examples were compiled on a Sun Solaris operating system. Since C-to-FORTRAN interfaces vary across platforms (operating system, hardware, compilers, etc.), you may need to modify the examples for your particular system.

Whether you need intermediate “glue” routines for the interface depends on your operating system. As a first step in building such an interface, we advise you to study your system documentation about C-to-FORTRAN interfaces. In that context, this section lists a few considerations particular to ILOG CPLEX in building a FORTRAN interface.

Case-Sensitivity

As you know, FORTRAN is a case-*in*sensitive language, whereas routines in the ILOG CPLEX Callable Library have names with mixed case. Most FORTRAN compilers have an option, such as the option `-U` on UNIX systems, that treats symbols in a case-sensitive way. We recommend that you use this option in any file that calls ILOG CPLEX Callable Library routines.

On some operating systems, such as DEC Alpha running Digital Unix, certain intrinsic FORTRAN functions must be in all upper case (that is, capital letters) for the compiler to accept those functions.

Underscore

On some systems, all FORTRAN external symbols are created with an underscore character (that is, _) added to the end of the symbol name. Some systems have an option to turn off this feature; for example, on DEC Alpha running Digital Unix, the option `-assume nounderscore` turns off the postpended underscore. If you are able to turn off those postpended underscores, you may not need other “glue” routines.

Six-Character Identifiers

FORTRAN 77 allows identifiers that are unique only up to six characters. However, in practice, most FORTRAN compilers allow you to exceed this limit. Since routines in the Callable Library have names greater than six characters, you need to verify whether your FORTRAN compiler enforces this limit or allows longer identifiers.

Call by Reference

By default, FORTRAN passes arguments by reference; that is, the *address* of a variable is passed to a routine, not its value. In contrast, many routines of the Callable Library require arguments passed by value. To accommodate those routines, most FORTRAN compilers have the VMS FORTRAN extension `%VAL()`. This operator used in calls to external functions or subroutines causes its argument to be passed by value (rather than by the default FORTRAN convention of passed by reference). For example, with that extension, you can call the routine `CPXprimopt()` with this FORTRAN statement:

```
status = CPXprimopt (%val(env), %val(lp))
```

Pointers

Certain ILOG CPLEX routines return a pointer to memory. In FORTRAN 77, such a pointer cannot be dereferenced; however, you can store its value in an appropriate integer type, and you can then pass it to other ILOG CPLEX routines. On most operating systems, the default integer type of four bytes is sufficient to hold pointer variables. On some systems, such as DEC Alpha, a variable of type `INTEGER*8` may be needed. Consult your system documentation to determine the appropriate integer type to hold variables that are C pointers.

Strings

When you pass strings to routines of the Callable Library, they expect C strings; that is, strings terminated by an ASCII `NULL` character, denoted \0 in C. Consequently, when you pass a FORTRAN string, you must add a terminating `NULL` character; you do so by means of the FORTRAN intrinsic function `CHAR(0)`.

C++ Interface

The ILOG CPLEX header file, `cplex.h`, includes the `extern` C statements necessary for use with C++. You include it directly in C++ source. The standard distribution of ILOG CPLEX includes examples of wrapper classes for C++ applications. The wrapper classes do not implement all Callable Library routines, but they are easy to extend for any routines you do need.

Managing Parameters from the Callable Library

Some ILOG CPLEX parameters assume values of type `double`; others assume values of type `int`; others are strings (that is, C-type `char*`). Consequently, in the Callable Library, there are *sets* of routines—one for `int`, one for `double`, one for `char*`—to access and to change parameters that control the ILOG CPLEX environment and guide optimization.

For example, the routine `CPXinfointparam()` shows you the default, the maximum, and the minimum values of a given parameter of type `int`, whereas the routine `CPXinfodblparam()` shows you the default, the maximum, and the minimum values of a given parameter of type `double`, and the routine `CPXinfostrparam()` shows you the default value of a given string parameter. Those three Callable Library routines observe the same conventions: they return `0` from a successful call and a nonzero value in case of error.

The routines `CPXinfointparam()` and `CPXinfodblparam()` expect five arguments:

- a pointer to the environment; that is, a pointer of type `CPXENVptr` returned by `CPXopenCPLEX()`;
- an indication of the parameter to check; this argument may be a symbolic constant, such as `CPX_PARAM_CLOCKTYPE`, or a reference number, such as `1006`; the symbolic constants and reference numbers of all ILOG CPLEX parameters are documented in the *ILOG CPLEX Reference Manual* and they are defined in the include file `cplex.h`.
- a pointer to a variable to hold the default value of the parameter;
- a pointer to a variable to hold the minimum value of the parameter;
- a pointer to a variable to hold the maximum value of the parameter.

The routine `CPXinfostrparam()` differs slightly in that it does not expect pointers to variables to hold the minimum and maximum values as those concepts do not apply to a string parameter.

To access the *current* value of a parameter that interests you from the Callable Library, use the routine `CPXgetintparam()` for parameters of type `int`, `CPXgetdblparam()` for parameters of type `double`, and `CPXgetstrparam()` for string parameters. These routines also expect arguments to indicate the environment, the parameter you want to check, and a pointer to a variable to hold that current value.

No doubt you have noticed in other chapters of this manual that you can *set* parameters from the Callable Library. There are, of course, routines in the Callable Library to set such parameters: one sets parameters of type `int`; another sets parameters of type `double`; another sets string parameters.

- `CPXsetintparam()` accepts arguments to indicate:
  - the environment; that is, a pointer of type `CPXENVptr` returned by `CPXopenCPLEX()`;
  - the parameter to set; this routine sets parameters of type `int`;
  - the value you want the parameter to assume.
- `CPXsetdblparam()` accepts arguments to indicate:
  - the environment; that is, a pointer of type `CPXENVptr` returned by `CPXopenCPLEX()`;
  - the parameter to set; this routine sets parameters of type `double`;
  - the value you want the parameter to assume.
- `CPXsetstrparam()` accepts arguments to indicate:
  - the environment; that is, a pointer of type `CPXENVptr` returned by `CPXopenCPLEX()`;
  - the parameter to set; this routine sets parameters of type `char*`;
  - the value you want the parameter to assume.

The *ILOG CPLEX Reference Manual* documents the type of each parameter (`int`, `double`, `char*`) along with the symbolic constant and reference number representing the parameter.

The routine `CPXsetdefaults()` *resets all parameters* (except the name of the log file) to their default values, including the ILOG CPLEX callback functions. This routine resets the callback functions to `NULL`. Like other Callable Library routines to manage parameters, this one accepts an argument indicating the environment, and it returns `0` for success or a nonzero value in case of error.

Example: Dietary Optimization

The optimization problem solved in this example is to compose a diet from a set of foods, so that the nutritional requirements are satisfied and the total cost is minimized. Example `diet.c` illustrates:

- Creating a Model Row by Row
- Creating a Model Column by Column
- Solving the Model with CPXlpopt()

Problem Representation

The problem contains a set of foods, which are the modeling variables; a set of nutritional requirements to be satisfied, which are the constraints, and an objective of minimizing the total cost of the food. There are two ways to look at this problem:

- The problem can be modeled in a row-wise fashion, by entering the variables first and then adding the constraints on the variables and the objective function.
- The problem can be modeled in a column-wise fashion, by constructing a series of empty constraints and then inserting the variables into the constraints and the objective function.

The diet problem is equally suited for both kinds of modeling. In fact you can even mix both approaches in the same program: If a new food product, you can create a new variable for it, regardless of how the model was originally built. Similarly, is a new nutrient is discovered, you can add a new constraint for it.

Creating a Model Row by Row

You walk into the store and compile a list of foods that are offered. For each food, you store the price per unit and the amount they have in stock. For some foods that you particularly like, you also set a minimum amount you would like to use in your diet. Then for each of the foods you create a modeling variable to represent the quantity to be purchased for your diet.

Now you get a medical book and look up which nutrients are known and relevant for you. For each nutrient, you note the minimum and maximum amount that should be found in your diet. Also, you go through the list of foods and determine how much a food item will contribute for each nutrient. This gives you one constraint per nutrient, which can naturally be represented as a range constraint

```
nutrmin[i] <= sum_j (nutrper[i][j] * buy[j]) <= nutrmax[i]
```

where `i` represents the number of the nutrient under consideration, `nutrmin[i]` and `nutrmax[i]` the minimum and maximum amount of nutrient `i` and `nutrper[i][j]` the amount of nutrient `i` in food `j`. Finally, you specify your objective function

```
sense = sum_j (cost[j] * buy[j])
```

This way to create the model is shown in function `populatebyrow` in example `diet.c`.

Creating a Model Column by Column

You start with the medical book where you compile the list of nutrients that you want to ensure are properly represented in your diet. For each of the nutrients you create an empty constraint

`nutrmin[i] <= ... <= nutrmax[i]`

where `...` is left to be filled once you walk into your store. Also you setup the objective function to minimize the cost. We will refer to constraint `i` as `rng[i]` and to the objective as `cost`.

Now you walk into the store and, for each food, you check its price and nutritional content. With this data you create a variable representing the amount you want to buy of the food type and install it in the objective function and constraints. That is you create the following column:

```
cost(foodCost[j]) "+" "sum_i" (rng[i](nutrper[i][j]))
```

where the notation "`+`" and "`sum`" indicates that you "add" the new variable `j` to the objective `cost` and constraints range[`i`]. The value in parenthesis is the linear coefficient that is used for the new variable). We chose this notation for its similarity to the syntax actually used in the Callable Library as demonstrated in function `populatebycolumn` in example `diet.c`.

Program Description

All definitions needed for a CPLEX Callable Library program are imported by including file `<ilcplex/cplex.h>` at the beginning of the program. After a number of lines that establish the calling sequences for the routines that to be used, the program named `main` begins by checking for correct command line arguments, printing a usage reminder and exiting in case of errors.

Next, the data defining the problem are read from a file specified in the command line at run time. The details of this are handled in the routine `readdata`. In this file, cost, lower bound, and upper bound are specified for each type of food; then minimum and maximum levels of several nutrients needed in the diet are specified; finally, a table giving levels of each nutrient found in each unit of food is given. The result of a successful call to this routine is two variables `&nfoods` and `&nnutr` containing the number of foods and nutrients in the data file, arrays `&cost`, `&lb`, `&ub` containing the information on the foods, arrays `&nutrmin`, `&nutrmax` containing nutritional requirements for the proposed diet, and array `&nutrper` containing the nutritional value of the foods.

Preparations to build and solve the model with CPLEX begin with the call to `CPXopenCPLEX()`. This establishes a CPLEX environment in which to contain the LP problem, and succeeds only if a valid CPLEX license is found.

After some calls to set parameters, one to control the output that comes to the user's terminal, and the other to turn on data checking for debugging purposes, a problem object is initialized through the call to `CPXcreateprob()`. This call returns a pointer to an empty problem object, which now can be populated with data.

Two alternative approaches to filling this problem object are implemented in this program, `populatebyrow()` and `populatebycolumn()`, and which one is executed is determined at run time by a calling parameter on the command line. The routine `populatebyrow()` operates by first defining all the columns through a call to `CPXnewcols()` and then repeatedly calls `CPXaddrows()` to enter the data of the constraints. The routine `populatebycolumn()` takes the complementary approach of establishing all the rows first with a call to `CPXnewrows()` and then sequentially adds the column data by calls to `CPXaddcols()`.

Solving the Model with CPXlpopt()

The model is at this point ready to be solved, and this is accomplished through the call to `CPXlpopt()`, which by default uses the dual simplex optimizer.

After this, the program finishes by making a call to `CPXsolution()` to obtain the values for each variable in this optimal solution, printing these values, and writing the problem to a disk file (for possible evaluation by the user) via the call to `CPXwriteprob()`. It then terminates after freeing all the arrays that have been allocated along the way.

Complete Program

The complete program, `diet.c`, appears here or online in the standard distribution.

```
#include <ilcplex/cplex.h>

/* Bring in the declarations for the string functions */

#include <stdlib.h>
#include <string.h>

/* Include declaration for functions at end of program */

#ifndef  CPX_PROTOTYPE_MIN

static int
   readarray         (FILE *in, int *num_p, double **data_p),
   readdata          (char* file,
                      int *nfoods_p, double **cost_p, double **lb_p, double
**ub_p,
                      int *nnutr_p, double **nutrmin_p, double **nutrmax_p,
                      double ***nutrper_p),
   populatebyrow     (CPXENVptr env, CPXLPptr lp,
                      int nfoods, double *cost, double *lb, double *ub,
                        int nnutr, double *nutrmin, double *nutrmax,
                        double **nutrper),
   populatebycolumn  (CPXENVptr env, CPXLPptr lp,
                        int nfoods, double *cost, double *lb, double *ub,
                        int nnutr, double *nutrmin, double *nutrmax,
                        double **nutrper);

static void
   free_and_null     (char **ptr),
   usage             (char *progname);

#else

static int
   readarray         (),
   readdata          (),
   populatebyrow     (),
   populatebycolumn  ();

static void
   free_and_null     (),
   usage             ();

#endif


#ifndef  CPX_PROTOTYPE_MIN
int
main (int argc, char **argv)
#else
int
main (argc, argv)
int  argc;
char **argv;
#endif
{
   int status = 0;

   int    nfoods;
   int    nnutr;
   double *cost     = NULL;
   double *lb       = NULL;
   double *ub       = NULL;
   double *nutrmin  = NULL;
   double *nutrmax  = NULL;
   double **nutrper = NULL;

   double *x = NULL;
   double objval;
   int    solstat;

   /* Declare and allocate space for the variables and arrays where we
      will store the optimization results including the status, objective
      value, variable values, dual values, row slacks and variable
      reduced costs. */

   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
```

```
int           i, j;

/* Check the command line arguments */

if (( argc != 3 )                         ||
    ( argv[1][0] != ‘-’ )                 ||
    ( strchr (“rc”, argv[1][1]) == NULL )  ) {
   usage (argv[0]);
   goto TERMINATE;
}

status = readdata(argv[2], &nfoods, &cost, &lb, &ub,
                  &nnutr, &nutrmin, &nutrmax, &nutrper);
if ( status ) goto TERMINATE;


/* Initialize the CPLEX environment */

env = CPXopenCPLEX (&status);

/* If an error occurs, the status value indicates the reason for
   failure.  A call to CPXgeterrorstring will produce the text of
   the error message.  Note that CPXopenCPLEX produces no output,
   so the only way to see the cause of the error is to use
   CPXgeterrorstring.  For other CPLEX routines, the errors will
   be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

if ( env == NULL ) {
   char  errmsg[1024];
   fprintf (stderr, “Could not open CPLEX environment.\n”);
   CPXgeterrorstring (env, status, errmsg);
   fprintf (stderr, “%s”, errmsg);
   goto TERMINATE;
}

/* Turn on output to the screen */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
if ( status ) {
   fprintf (stderr,
            “Failure to turn on screen indicator, error %d.\n”, status);
   goto TERMINATE;
}

/* Turn on data checking */

status = CPXsetintparam (env, CPX_PARAM_DATACHECK, CPX_ON);
if ( status ) {
   fprintf (stderr,
            “Failure to turn on data checking, error %d.\n”, status);
   goto TERMINATE;
}

/* Create the problem. */

lp = CPXcreateprob (env, &status, “diet”);

/* A returned pointer of NULL may mean that not enough memory
```

```
      was available or there was some other problem.  In the case of
      failure, an error message will have been written to the error
      channel from inside CPLEX.  In this example, the setting of
      the parameter CPX_PARAM_SCRIND causes the error message to
      appear on stdout.  */

   if ( lp == NULL ) {
      fprintf (stderr, “Failed to create LP.\n”);
      goto TERMINATE;
   }

   /* Now populate the problem with the data.  For building large
      problems, consider setting the row, column and nonzero growth
      parameters before performing this task. */

   switch (argv[1][1]) {
      case ‘r’:
         status = populatebyrow (env, lp, nfoods, cost, lb, ub,
                                 nnutr, nutrmin, nutrmax, nutrper);
         break;
      case ‘c’:
         status = populatebycolumn (env, lp, nfoods, cost, lb, ub,
                                    nnutr, nutrmin, nutrmax, nutrper);
         break;
   }

   if ( status ) {
      fprintf (stderr, “Failed to populate problem.\n”);
      goto TERMINATE;
   }

   /* Optimize the problem and obtain solution. */

   status = CPXlpopt (env, lp);
   if ( status ) {
      fprintf (stderr, “Failed to optimize LP.\n”);
      goto TERMINATE;
   }

   x = (double *) malloc (nfoods * sizeof(double));
   if ( x == NULL ) {
      status = CPXERR_NO_MEMORY;
      fprintf (stderr, “Could not allocate memory for solution.\n”);
      goto TERMINATE;
   }

   status = CPXsolution (env, lp, &solstat, &objval, x, NULL, NULL, NULL);
   if ( status ) {
      fprintf (stderr, “Failed to obtain solution.\n”);
      goto TERMINATE;
   }

   /* Write the output to the screen. */

   printf (“\nSolution status = %d\n”, solstat);
   printf (“Solution value  = %f\n\n”, objval);

   for (j = 0; j < nfoods; j++)
```

```
      printf (“Food %d:  Buy = %10f\n”, j, x[j]);

   /* Finally, write a copy of the problem to a file. */

   status = CPXwriteprob (env, lp, “diet.lp”, NULL);
   if ( status ) {
      fprintf (stderr, “Failed to write LP to disk.\n”);
      goto TERMINATE;
   }

TERMINATE:

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, “CPXfreeprob failed, error code %d.\n”, status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status > 0 ) {
         char  errmsg[1024];
         fprintf (stderr, “Could not close CPLEX environment.\n”);
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, “%s”, errmsg);
      }
   }

   if ( nutrper != NULL ) {
      for (i = 0; i < nnutr; ++i) {
         free_and_null ((char **) &(nutrper[i]));
      }
   }
   free_and_null ((char **) &nutrper);
   free_and_null ((char **) &cost);
   free_and_null ((char **) &cost);
   free_and_null ((char **) &lb);
   free_and_null ((char **) &ub);
   free_and_null ((char **) &nutrmin);
   free_and_null ((char **) &nutrmax);
   free_and_null ((char **) &x);

   return (status);

}  /* END main */
```

```
#ifndef  CPX_PROTOTYPE_MIN
static int
populatebyrow (CPXENVptr env, CPXLPptr lp,
               int nfoods, double *cost, double *lb, double *ub,
               int nnutr, double *nutrmin, double *nutrmax,
               double **nutrper)
#else
static int
populatebyrow (env, lp)
CPXENVptr  env;
CPXLPptr   lp;
int        nfoods;
double     *cost;
double     *lb;
double     *ub;
int        nnutr;
double     *nutrmin;
double     *nutrmax;
double     **nutrper;
#endif
{
   int status = 0;

   int zero = 0;
   int *ind = NULL;
   int i, j;

   ind = (int*) malloc(nfoods * sizeof(int));
   if ( ind == NULL ) {
      status = CPXERR_NO_MEMORY;
      goto TERMINATE;
   }
   for (j = 0; j < nfoods; j++) ind[j] = j;

   status = CPXnewcols (env, lp, nfoods, cost, lb, ub, NULL, NULL);
   if ( status )  goto TERMINATE;

   for (i = 0; i < nnutr; i++) {
      double rng  = nutrmax[i] - nutrmin[i];

      status = CPXaddrows (env, lp, 0, 1, nfoods, nutrmin+i, “R”,
                           &zero, ind, nutrper[i], NULL, NULL);
      if ( status )  goto TERMINATE;

      status = CPXchgrngval (env, lp, 1, &i, &rng);
      if ( status )  goto TERMINATE;
   }

TERMINATE:

   free_and_null ((char **)&ind);

   return (status);

}  /* END populatebyrow */
```

```
/* To populate by column, we first create the rows, and then add the
   columns.  */

#ifndef  CPX_PROTOTYPE_MIN
static int
populatebycolumn (CPXENVptr env, CPXLPptr lp,
                  int nfoods, double *cost, double *lb, double *ub,
                  int nnutr, double *nutrmin, double *nutrmax,
                  double **nutrper)
#else
static int
populatebycolumn (env, lp)
CPXENVptr  env;
CPXLPptr   lp;
int        nfoods;
double     *cost;
double     *lb;
double     *ub;
int        nnutr;
double     *nutrmin;
double     *nutrmax;
double     **nutrper;
#endif
{
   int status = 0;

   int i, j;

   int    zero    = 0;
   int    *ind    = NULL;
   double *val    = NULL;
   char   *sense  = NULL;
   double *rngval = NULL;

   sense = (char*)malloc(nnutr * sizeof(char));
   if ( sense == NULL ) {
      status = CPXERR_NO_MEMORY;
      goto TERMINATE;
   }
   for (i = 0; i < nnutr; i++) sense[i] = ‘R’;

   val = (double*)malloc(nnutr * sizeof(double));
   if ( val == NULL ) {
      status = CPXERR_NO_MEMORY;
      goto TERMINATE;
   }

   rngval = (double*)malloc(nnutr * sizeof(double));
   if ( rngval == NULL ) {
      status = CPXERR_NO_MEMORY;
      goto TERMINATE;
   }
   for (i = 0; i < nnutr; i++) rngval[i] = nutrmax[i] - nutrmin[i];

   ind = (int*) malloc(nfoods * sizeof(int));
   if ( ind == NULL ) {
      status = CPXERR_NO_MEMORY;
      goto TERMINATE;
   }
   for (i = 0; i < nnutr; i++) ind[i] = i;

   status = CPXnewrows (env, lp, nnutr, nutrmin, sense, rngval, NULL);
   if ( status )  goto TERMINATE;

   for (j = 0; j < nfoods; ++j) {
      for (i = 0; i < nnutr; i++) val[i] = nutrper[i][j];

      status = CPXaddcols (env, lp, 1, nnutr, cost+j, &zero,
                           ind, val, lb+j, ub+j, NULL);
      if ( status )  goto TERMINATE;
   }

TERMINATE:

   free_and_null ((char **)&sense);
   free_and_null ((char **)&rngval);
   free_and_null ((char **)&ind);
   free_and_null ((char **)&val);

   return (status);

}  /* END populatebycolumn */


/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

#ifndef  CPX_PROTOTYPE_MIN
static void
free_and_null (char **ptr)
#else
static void
free_and_null (ptr)
char  **ptr;
#endif
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */



#ifndef  CPX_PROTOTYPE_MIN
static void
usage (char *progname)
#else
static void
usage (progname)
char *progname;
#endif
{
   fprintf (stderr,”Usage: %s -X <datafile>\n”, progname);
   fprintf (stderr,”   where X is one of the following options: \n”);
   fprintf (stderr,”      r          generate problem by row\n”);
   fprintf (stderr,”      c          generate problem by column\n”);
   fprintf (stderr,” Exiting...\n”);
} /* END usage */


#ifndef  CPX_PROTOTYPE_MIN
static int
readarray (FILE *in, int *num_p, double **data_p)
#else
static int
readarray()
FILE   *in;
int    *num_p;
double **data_p;
#endif
{
   int  status = 0;
   int  max, num;
   char ch;

   num = 0;
   max = 10;

   *data_p = (double*)malloc(max * sizeof(double));
   if ( *data_p == NULL ) {
      status = CPXERR_NO_MEMORY;
      goto TERMINATE;
   }

   for (;;) {
      fscanf (in, “%c”, &ch);
      if ( ch == ‘\t’ ||
           ch == ‘\r’ ||
           ch == ‘ ‘  ||
           ch == ‘\n’   ) continue;
      if ( ch == ‘[‘ ) break;
      status = -1;
      goto TERMINATE;
   }

   for(;;) {
      int read;
      read = fscanf (in, “%lg”, (*data_p)+num);
      if ( read == 0 ) {
         status = -1;
         goto TERMINATE;
      }
      num++;
      if ( num >= max ) {
         max *= 2;
         *data_p = (double*)realloc(*data_p, max * sizeof(double));
         if ( *data_p == NULL ) {
            status = CPXERR_NO_MEMORY;
            goto TERMINATE;
         }
      }
      do {
         fscanf (in, “%c”, &ch);
      } while (ch == ‘ ‘ || ch == ‘\n’ || ch == ‘\t’  || ch == ‘\r’);
      if ( ch == ‘]’ ) break;
      else if ( ch != ‘,’ ) {
         status = -1;
         goto TERMINATE;
      }
   }

   *num_p = num;

TERMINATE:

   return (status);

} /* END readarray */


#ifndef  CPX_PROTOTYPE_MIN
static int
readdata (char* file,
          int *nfoods_p, double **cost_p, double **lb_p, double **ub_p,
          int *nnutr_p, double **nutrmin_p, double **nutrmax_p,
          double ***nutrper_p)
#else
static int
readdata ()
char   *file;
int    *nfoods_p;
double **cost_p;
double **lb_p;
double **ub_p;
int    *nnutr_p;
double **nutrmin_p;
double **nutrmax_p;
double ***nutrper_p;
#endif
{
   int status = 0;

   int ncost, nlb, nub;
   int nmin, nmax;

   int  i, n;
   char ch;
   FILE *in = NULL;

   in = fopen(file, “r”);
   if ( in == NULL ) {
      status = -1;
      goto TERMINATE;
   }

   if ( (status = readarray(in, &ncost, cost_p)) ) goto TERMINATE;
   if ( (status = readarray(in, &nlb,   lb_p))   ) goto TERMINATE;
   if ( (status = readarray(in, &nub,   ub_p))   ) goto TERMINATE;
   if ( ncost != nlb  ||  ncost != nub ) {
      status = -1;
      goto TERMINATE;
   }
   *nfoods_p = ncost;

   if ( (status = readarray(in, &nmin, nutrmin_p)) ) goto TERMINATE;
   if ( (status = readarray(in, &nmax, nutrmax_p)) ) goto TERMINATE;
   if ( nmax != nmin ) {
      status = -1;
      goto TERMINATE;
   }
   *nnutr_p = nmin;

   *nutrper_p = (double**)malloc(nmin * sizeof(double*));
   if ( *nutrper_p == NULL ) {
      status = CPXERR_NO_MEMORY;
      goto TERMINATE;
   }

   for (;;) {
      fscanf (in, “%c”, &ch);
      if ( ch == ‘\t’ ||
           ch == ‘\r’ ||
           ch == ‘ ‘  ||
           ch == ‘\n’   ) continue;
      if ( ch == ‘[‘ ) break;
      status = -1;
      goto TERMINATE;
   }
   for ( i = 0; i < nmin; i++ ) {
      if ( (status = readarray(in, &n, (*nutrper_p)+i)) ) goto TERMINATE;
      if ( n != ncost ) {
         status = -1;
         goto TERMINATE;
      }
      fscanf (in, “%c”, &ch);
      if ( i < nmin-1  &&  ch != ‘,’ ) {
         status = -1;
         goto TERMINATE;
      }
   }
   if ( ch != ‘]’ ) {
      status = -1;
      goto TERMINATE;
   }


TERMINATE:

   return (status);

} /* END readdata */
```

C H A P T E R

3

Further Programming Considerations

This chapter offers suggestions for improving application development and debugging completed applications. It includes sections on:

◆ Tips for Successful Application Development

◆ Using the Interactive Optimizer for Debugging

◆ Eliminating Common Programming Errors

Tips for Successful Application Development

In the previous chapters, we indicated briefly the minimal steps to use the Component Libraries in an application. This section offers guidelines for successfully developing an application that exploits the ILOG CPLEX Component Libraries according to those steps. These guidelines aim to help you minimize development time and maximize application performance.

Prototype the Model

We strongly recommend that you begin by creating a small-scale version of the model for your problem. (There are modeling languages, such as ILOG OPL, that may be helpful to you for this task.) This prototype model can serve as a test-bed for your application and a point of reference during development.

Identify Routines to Use

If you decompose your application into manageable components, you can more easily identify the tools you must complete the application. Part of this decomposition consists of determining which methods or routines from the ILOG CPLEX Component Libraries your application will call. Such a decomposition will assist you in testing for completeness; it may also help you isolate troublesome areas of the application during development; and it will aid you in measuring how much work is already done and how much remains.

Test Interactively

The Interactive Optimizer in ILOG CPLEX (introduced in the manual *ILOG CPLEX Getting Started*) offers a reliable means to test the ILOG CPLEX component of your application interactively, particularly if you have prototyped your problem model. Interactive testing through the Interactive Optimizer can also help you identify precisely which methods or routines from the Component Libraries your application needs. Additionally, interactive testing early in development may also uncover any flaws in procedural logic before they entail costly coding efforts.

Most importantly, optimization commands in the Interactive Optimizer perform exactly like optimization routines in the Component Libraries. In other words, the command `primopt` works just the same way as the method `IloCplex::setRootAlgorithm(IloCplex::Primal)` and the routine `CPXprimopt()`; likewise, the command `tranopt` works like the method `IloCplex::setRootAlgorithm(IloCplex::Dual)` and the routine `CPXdualopt()`; `netopt` works like `IloCplex::setRootAlgorithm(IloCplex::Barrier)` and `CPXhybnetopt()`, and so forth. Consequently, any discrepancy between the Interactive Optimizer and the Component Libraries routines with respect to the solutions found, memory used, or time taken indicates a problem in the logic of the application calling the routines.

Assemble Data Efficiently

As we indicated in previous chapters, ILOG CPLEX offers several ways of putting data into your problem or (more formally) populating the problem object. You must decide which approach is best adapted to your application, based on your knowledge of the problem data and application specifications. These considerations may enter into your decision:

◆ If your Callable Library application builds the arrays of the problem in memory and then calls `CPXcopylp()`, it avoids time-consuming reads from disk files.

◆ In the Callable Library, using the routines `CPXnewcols()`, `CPXnewrows()`, `CPXaddcols()`, `CPXaddrows()`, and `CPXchgcoeflist()` may help you build modular code that will be more easily modified and maintained than code that assembles all problem data in one step.

◆ An application that reads an MPS or LP file may reduce the coding effort but, on the other hand, may increase runtime and disk space requirements.

  Keep in mind that if an application using the ILOG CPLEX Component Libraries reads an MPS or LP file, then some other program must generate that formatted file. The data structures used to generate the file can almost certainly be used directly to build the problem-populating arrays for `CPXcopylp()` or `CPXaddrows()`—a choice resulting in less coding and a faster, more efficient application.

  In short, formatted files are useful for prototyping your application. For production purposes, assembly of data arrays in memory may be a better enhancement.

Test Data

CPLEX provides the `CPX_PARAM_DATACHECK` parameter to check the correctness of data used in the `CPXcopy...(), CPXread...(), and CPXchg...()` functions. When this parameter is set, CPLEX will perform extra checks to determine that array arguments contain valid values, such as indices within range, no duplicate entries, valid row sense indicators and valid numerical values. These checks can be very useful during development, but are probably too costly for deployed applications. The checks are similar to but not as extensive as those performed by the `CPXcheck...()` functions. When the parameter is not set (the default), only simple error checks are performed, for example, checking for the existence of the environment.

Choose an Optimizer

After you have instantiated and populated a problem object, you solve it by calling one of the optimizers available in the ILOG CPLEX Component Libraries. Your choice of optimizer depends on the type of problem:

◆ If the problem is a linear program, use the linear optimizer.

◆ If the linear program includes a large embedded network, consider using the network optimizer.

◆ If the problem includes integer variables (MIP), use the branch & cut algorithm (implemented in the separately licensed ILOG CPLEX Mixed Integer Optimizer).

◆ If the problem is a convex quadratic program (QP), use the primal-dual barrier method (implemented in the separately licensed ILOG CPLEX Barrier Optimizer).

In ILOG CPLEX, there are many possible parameter settings for each optimizer. Generally, the default parameter settings are best for linear programming problems, but Chapter 4, *Solving Linear Programming Problems*, offers more detail about improving performance with respect to LP problems. Integer programming problems are more sensitive to specific parameter settings, so you may need to experiment with them, as suggested in Chapter 5, *Solving Mixed Integer Programming Problems*.

In either case, the Interactive Optimizer in ILOG CPLEX lets you try different parameter settings and different optimizers to determine the best optimization procedure for your particular application. From what you learn by experimenting with commands in the Interactive Optimizer, you can more readily choose which method or routine from the Component Libraries to call in your application.

Program with a View toward Maintenance and Modifications

Good programming practices save development time and make an application easier to understand and modify. *Tips for Successful Application Development* on page 85 describes our programming conventions in developing ILOG CPLEX. In addition, we recommend the following programming practices.

Comment Your Code

Comments, written in mixed upper- and lower-case, will prove useful to you at a later date when you stare at code written months ago and try to figure out what it does. They will also prove useful to our staff, should you need to send us your application for technical support.

Write Readable Code

Follow conventional formatting practices so that your code will be easier to read, both for you and for others. Use fewer than 80 characters per line. Put each statement on a separate line. Use white space (for example, space, blank lines, tabs) to distinguish logical blocks of code. Display compound loops with clearly indented bodies. Display `if` statements like combs; that is, we recommend that you align `if` and `else` in the same column and then indent the corresponding block. Likewise, we recommend that you indent the body of compound statements, loops, and other structures distinctly from their corresponding headers and closing brackets. Use uniform indentation (for example, three to five spaces). Put at least one space before and after each relational operator, as well as before and after each binary plus (`+`) and minus (`-`). Use space as you do in normal English.

Avoid Side-Effects

We recommend that you minimize side-effects by avoiding expressions that produce internal effects. In C, for example, try to avoid expressions of this form:

```
a = c + (d =  e*f);  /* A BAD IDEA */
```

where the expression assigns the values of `d` and `a`.

Don’t Change Argument Values

A user-defined function should not change the values of its arguments. Do not use an argument to a function on the left-hand side of an assignment statement in that function. Since C and C++ pass arguments by value, treat the arguments strictly as values; do not change them inside a function.

Declare the Type of Return Values

Always declare the return type of functions explicitly. Though C has a “historical tradition” of making the default return type of all functions `int`, we recommend that you explicitly declare the return type of functions that return a value, and use `void` for procedures that do not return a value.

Manage the Flow of Your Code

Use only one `return` statement in any function. Limit your use of `break` statements to the inside of `switch` statements. In C, do not use `continue` statements and limit your use of `goto` statements to exit conditions that branch to the end of a function. Handle error conditions in C++ with a `try`/`catch` block and in C with a `goto` statement that transfers control to the end of the function so that your functions have only one exit point.

In other words, control the flow of your functions so that each block has one entry point and one exit point. This “one way in, one way out” rule makes code easier to read and debug.

Localize Variables

Avoid global variables at all costs. Code that exploits global variables invariably produces side-effects which in turn make the code harder to debug. Global variables also set up peculiar reactions that make it difficult to include your code successfully within other applications. Also global variables preclude multithreading unless you invoke locking techniques. As an alternative to global variables, pass arguments down from one function to another.

Name Your Constants

Scalars—both numbers and characters—that remain constant throughout your application should be named. For example, if your application includes a value such as 1000, create a constant with the `#define` statement to name it. If the value ever changes in the future, its occurrences will be easy to find and modify as a named constant.

Choose Clarity First, Efficiency Later

Code first for clarity. Get your code working accurately first so that you maintain a good understanding of what it is doing. Then, once it works correctly, look for opportunities to improve performance.

Debug Effectively

Using Diagnostic Routines for Debugging on page 66, contains tips and guidelines for debugging an application that uses the ILOG CPLEX Callable Library. In that context, we recommend using a symbolic debugger as well as other widely available development tools to produce error-free code.

Test Correctness, Test Performance

Even a program that has been carefully debugged so that it runs correctly may still contain errors or “features” that inhibit its performance with respect to execution speed, memory use, and so forth. Just as the ILOG CPLEX Interactive Optimizer can aid in your tests for correctness, it can also help you improve performance. It uses the same routines as the Component Libraries; consequently, it requires the same amount of time to solve a problem created by a callable-library application. We recommend that you use `CPXwriteprob()`, specifying a file type of SAV, to create a binary representation of the problem object of your application. Then read that representation into the Interactive Optimizer, and solve it there. If your application sets parameters, use the same settings in the Interactive Optimizer. If you find that your application takes significantly longer to solve the problem than does the Interactive Optimizer, then you can probably improve the performance of your application. In such a case, look closely at issues like memory fragmentation, unnecessary compiler options, inappropriate linker options, and programming practices that slow the application without causing incorrect results (such as operations within a loop that should be outside the loop).

Using the Interactive Optimizer for Debugging

The ILOG CPLEX Interactive Optimizer distributed with the Component Libraries offers a way to see what is going on within the ILOG CPLEX-part of your application when you observe peculiar behavior in your optimization application. The commands of the Interactive Optimizer correspond exactly to routines of the Component Libraries, so anomalies due to the ILOG CPLEX-part of your application will manifest themselves in the Interactive Optimizer as well, and contrariwise, if the Interactive Optimizer behaves appropriately on your problem, you can be reasonably sure that routines you call in your application from the Component Libraries work in the same appropriate way.

The first step in using the Interactive Optimizer for debugging is to write a version of the problem from the application into a formatted file that can then be loaded into the Interactive Optimizer. To do so, insert a call to the method `IloCplex::exportModel()` or to the routine `CPXwriteprob()` into your application. Use that call to create a file, whether an LP, SAV, or MPS formatted problem file. (*Understanding File Formats* on page 264 briefly describes these file formats.) Then read that file into the Interactive Optimizer and optimize the problem there.

Note that MPS, LP and SAV files have differences that influence how to interpret the results of the Interactive Optimizer for debugging. SAV files contain the exact binary representation of the problem as it appears in your program, while MPS and LP files are text files containing possibly less precision for numeric data. And, unless every variable appears on the objective function, CPLEX will probably order the variables differently when it reads the problem from an LP file than from an MPS or SAV file. With this in mind, SAV files are the most useful for debugging using the Interactive Optimizer, followed by MPS files, then finally LP files, in terms of the change in behavior you might see by use of explicit files. On the other hand, LP files are often quite helpful when you want to examine the problem, more so than as input for the Interactive Optimizer. Furthermore, try solving both the SAV and MPS files of the same problem using the Interactive Optimizer. Different results may provide additional insight into the source of the difficulty. In particular, use the following guidelines with respect to reproducing your program's behavior in the Interactive Optimizer.

1. If you can reproduce the behavior with a SAV file, but not with an MPS file, this suggests corruption or errors in the problem data arrays. Use the diagnostic routines in the source file check.c to track down the problem.

2. If you can reproduce the behavior in neither the SAV file nor the MPS file, the most likely cause of the problem is that your program has some sort of memory error. Memory debugging tools such as Purify or Insure will usually find such problems quickly.

3. When solving a problem in MPS or LP format, if the Interactive Optimizer issues a message about a segmentation fault or similar ungraceful interruption and exits, contact CPLEX technical support to arrange for transferring the problem file. The Interactive Optimizer should never exit with a system interrupt when solving a problem from a text file, even if the program that created the file has errors. Such cases are extremely rare.

If the peculiar behavior that you observed in your application persists in the Interactive Optimizer, then you must examine the LP or MPS or SAV problem file to determine whether the problem file actually defines the problem you intended. If it does not define the problem you intended to optimize, then the problem is being passed incorrectly from your application to ILOG CPLEX, so you need to look at that part of your application.

Make sure the problem statistics and matrix coefficients indicated by the Interactive Optimizer match the ones for the intended model in your application. Use the Interactive Optimizer command `display problem stats` to verify that the size of the problem, the sense of the constraints, and the types of variables match your expectations. For example, if your model is supposed to contain only general integer variables, but the Interactive Optimizer indicates the presence of binary variables, check the type variable passed to the constructor of the variable (Concert Technology Library) or check the specification of the `ctype` array and the routine `CPXcopyctype()` (Callable Library). You can also examine the matrix, objective, and right-hand side coefficients in an LP or MPS file to see if they are consistent with the values you expect in the model.

Eliminating Common Programming Errors

We hope this section serves as a checklist to help you eliminate common pitfalls from your application.

Check Your Include Files

Make sure that the header file `ilocplex.h` (Concert Technology Library) or `cplex.h` (Callable Library) is included at the top of your application source file. If that file is not included, then compile-time, linking, or runtime errors may occur.

Clean House and Try Again

Remove all object files, recompile, and relink your application.

Read Your Messages

ILOG CPLEX detects many different kinds of errors and generates exception, warnings, or error messages about them.

To query exceptions in the Concert Technology Library, use the methods:

```
IloInt IloCplex::Exception::getStatus() const;
const char* IloException::getMessage() const;
```

To view warnings and error messages in the Callable Library, you must direct them either to your screen or to a log file.

- ◆ To direct all messages to your screen, use the routine `CPXsetintparam()` to set the parameter `CPX_PARAM_SCRIND`.
- ◆ To direct all messages to a log file, use the routine `CPXsetlogfile()`.

Check Return Values

Most methods and routines of the Component Libraries return a value that indicates whether the routine failed, where it failed, and why it failed. This return value can help you isolate the point in your application where an error occurs.

If a return value indicates failure, always check whether sufficient memory is available.

Beware of Numbering Conventions

If you delete a portion of a problem, ILOG CPLEX changes not only the dimensions but also the indices of the problem. If your application continues to use the former dimensions and indices, errors will occur. Therefore, in parts of your application that delete portions of the problem, look carefully at how dimensions and indices are represented.

Make Local Variables Temporarily Global

If you are having difficulty tracking down the source of an anomaly in the heap, try making certain local variables *temporarily* global. This debugging trick may prove useful after your application reads in a problem file or modifies a problem object. If application behavior changes when you change a local variable to global, then you may get from it a better idea of the source of the anomaly.

Solve the Problem You Intended

Your application may inadvertently alter the problem and thus produce unexpected results. To check whether your application is solving the problem you intended, use the Interactive Optimizer, as we suggest on page 90, and the diagnostic routines, as described on page 66.

You should not ignore any ILOG CPLEX warning message in this situation either, so read your messages, as we suggest on page 92.

If you are working in the Interactive Optimizer, we also suggest that you use the command `display problem stats` to check the problem dimensions.

Special Considerations for Fortran

Check row and column indices. Fortran conventionally numbers from one (`1`), whereas C and C++ number from zero (`0`). This difference in numbering conventions can lead to unexpected results with regard to row and column indices when your application modifies a problem or exercises query routines.

We strongly recommend that you use the Fortran declaration `IMPLICIT NONE` to help you detect any unintended type conversions, as such inadvertent conversions frequently lead to strange application behavior.

Tell Us

Finally, if your problem remains unsolved by ILOG CPLEX, or if you believe you have discovered a bug in ILOG CPLEX, we would appreciate hearing from you about it.

C H A P T E R

4

Solving Linear Programming Problems

This chapter tells you more about solving linear programs with ILOG CPLEX using the LP optimizers. It contains sections on:

- Choosing an Optimizer for Your LP Problem
- Tuning LP Performance
- Diagnosing Performance Problems
- Diagnosing LP Infeasibility
- Example: Using a Starting Basis in an LP Problem
- Solving LP Problems with the Barrier Optimizer
- Interpreting the Barrier Log File
- Understanding Solution Quality from the Barrier LP Optimizer
- Overcoming Numerical Difficulties
- Diagnosing Barrier Optimizer Infeasibility

Choosing an Optimizer for Your LP Problem

As we explain in *Using the Callable Library in an Application* on page 57, to exploit ILOG CPLEX in your own application, you must first create a ILOG CPLEX environment, instantiate a problem object, and populate the problem object with data. As your next step, you call a ILOG CPLEX optimizer. ILOG CPLEX offers several different optimizers for linear programming problems. All of these optimizers are available to you in three forms, as you see in Table 4.1: as commands for you to issue in the Interactive Optimizer, as parameters to select in the Concert Technology Library, and as routines to call from the Callable Library in your own application.

Table 4.1 *Optimizers for Linear Programming (LP) Problems*

Optimizer	**Interactive Command**	**Concert Technology Library Parameter**	**Callable Library Routine**
automatically chosen	`optimize`	`cplex.solve()`	`CPXlpopt()`
primal simplex	`primopt`	`cplex.setRootAlgorithm(IloCplex::Primal)`	`CPXprimopt()`
dual simplex	`tranopt`	`cplex.setRootAlgorithm(IloCplex::Dual)`	`CPXdualopt()`
network	`netopt`	`cplex.setRootAlgorithm(IloCplex::NetworkDual)`	`CPXhybnetopt()`
primal-dual barrier	`baropt`	`cplex.setRootAlgorithm(IloCplex::BarrierPrimal)`	`CPXhybbaropt()`

Automatic Selection of Best Optimizer

If you are unfamiliar with the relationship between problem characteristics and optimizer speed, you may prefer to let CPLEX determine the best algorithm to use to optimize your problem. Most models are well solved by the default optimizer selected by calling `optimize` / `cplex.solve()` / `CPXlpopt()`. We recommend using this option unless you wish to tune performance.

Under defaults, CPLEX solves an LP model using the dual simplex method. The primal simplex method is available as an alternative optimizer and can be faster on some models. To determine whether your problem contains a network, try the network optimizer as well. Additionally, if you are licensed to use it, we suggest that you try the primal-dual logarithmic barrier optimizer (that is, the ILOG CPLEX Barrier Optimizer); it is applicable to very large, sparse problems, particularly those with a block-matrix structure. The following sections say more about each linear optimizer.

Dual Simplex Optimizer

If you are familiar with linear programming theory, then you recall that a linear programming problem can be stated in primal or dual form, and an optimal solution (if one exists) of the dual has a direct relationship to an optimal solution of the primal model. CPLEX's Dual Simplex Optimizer makes use of this relationship, but still reports the solution in terms of the primal model. Recent computational advances in the dual simplex method have made it the first choice for optimizing a linear programming problem. This is especially true for primal-degenerate problems with little variability in the right-hand side coefficients but significant variability in the cost coefficients.

Primal Simplex Optimizer

CPLEX's Primal Simplex Optimizer also can effectively solve a wide variety of linear programming problems with its default parameter settings. With recent advances in the dual simplex method, the primal simplex method is no longer the obvious choice for a first try at optimizing a linear programming problem. However, this method will sometimes work better on problems where the number of variables exceeds the number of constraints significantly, or on problems that exhibit little variability in the cost coefficients. Few problem exhibit poor numerical performance in both primal and dual form. Consequently, if you have a problem where numerical difficulties occur when you use the dual simplex optimizer, then consider using the primal simplex optimizer instead.

Network Optimizer

If a major part of your problem is structured as a network, then the ILOG CPLEX Network Optimizer may have a positive impact on performance. The ILOG CPLEX Network Optimizer recognizes a special class of linear programming problems with network structure. It uses highly efficient network algorithms on that part of the problem to find a solution from which it then constructs an advanced basis for the rest of your problem. From this advanced basis, ILOG CPLEX then iterates to find a solution to the full problem. Chapter 6, *Solving Network-Flow Problems* describes this optimizer in greater detail.

Primal-Dual Barrier Optimizer

The optional primal-dual ILOG CPLEX Barrier Optimizer requires a special license. It offers an approach completely different from the primal and dual simplex optimizers and from the network optimizer—an approach particularly efficient in large, sparse problems (for example, more than 1000 rows or columns, relatively few nonzeros per column). *Solving LP Problems with the Barrier Optimizer* on page 129 explains this optimizer in greater detail in the context of linear programming, and Chapter 7, *Solving Quadratic Programming Problems* covers this optimizer in the context of convex quadratic objective functions.

Tuning LP Performance

Each of the optimizers available in CPLEX is designed to solve most linear programming problems under its default parameter settings. However, characteristics of your particular problem may make performance tuning advantageous.

As a first step in tuning performance, try the different CPLEX optimizers, as we suggested in *Choosing an Optimizer for Your LP Problem* on page 96. The following sections describe other features of CPLEX to consider in tuning the performance of your application:

- Preprocessing: Presolver and Aggregator
- Preprocessing: Explicitly Solving the Dual
- Starting from an Advanced Basis
- Adjusting Parameters

Preprocessing: Presolver and Aggregator

By default, the preprocessing parameters of ILOG CPLEX are on. That is, ILOG CPLEX customarily preprocesses problems by simplifying constraints, reducing problem size, and eliminating redundancy. Its presolver tries to reduce the size of a problem by decreasing the number of rows and columns. Its aggregator tries to eliminate variables and rows through substitution. However, if your problem contains no redundancy nor other opportunities for simplification, then it will solve faster and it will save memory if you turn off the preprocessing in ILOG CPLEX.

- This command turns off preprocessing in the Interactive Optimizer: `set preprocessing presolve no`
- To turn off preprocessing when using the Component Libraries, set the parameter `IloCplex::PreInd or CPX_PARAM_PREIND`.

By default, ILOG CPLEX will not invoke the aggregator when the presolver is off.

Rarely, a preprocessed problem may prove more difficult than the original. In such cases, to improve performance, turn the presolver off or alternatively, specify a particular number of passes for the presolver to make through the model by setting the `numpass` parameter to a positive number.

Occasionally, the substitutions that the ILOG CPLEX aggregator makes will increase matrix density and thus make each iteration too expensive to be advantageous. In such cases, lower the preprocessing fill parameter; it limits substitutions to minimize the addition of nonzeros. ILOG CPLEX will make fewer substitutions as a consequence, and the resulting problem will be less dense.

To lower the preprocessing fill parameter:

- ◆ In the Interactive Optimizer, use the command `set preprocessing fill` with a lower value than its default value of 10.
- ◆ When using the Component Libraries, set the parameter `IloCplex::AggFill` or `CPX_PARAM_AGGFILL`.

By default, ILOG CPLEX applies its aggregator once when it is using the LP optimizers. For some problems, it may be worthwhile to apply the aggregator more than once. In those cases, set the preprocessing aggregator parameter to a positive integer value.

To apply the aggregator more than once:

- ◆ In the Interactive Optimizer, use the command, for example: `set preprocessing aggregator 2`.
- ◆ When using the Component Libraries, set the parameter `IloCplex::AggInd` or `CPX_PARAM_AGGIND`.

In cases where your model may be primal infeasible or unbounded (dual infeasible), it may be desirable to control the kinds of presolve reductions which CPLEX makes, in order to make your analysis of the outcome of optimization more certain. These reductions can be divided into two types: primal reductions and dual reductions. A reduction is primal if it doesn't depend on the objective function. A reduction is dual if it doesn't depend on the right hand side. By default, presolve performs both kinds of reductions.

Under the default, if the presolved model is infeasible, we know only that the original model is either infeasible or unbounded. But if presolve has performed only primal reductions and the presolved model is infeasible, then we have full assurance that the original model is also infeasible. Similarly if presolve has performed only dual reductions and the presolved model is unbounded, then the original model is verified as unbounded.

To control the dual reductions performed by presolve. In the:

- ◆ Interactive Optimizer, the command: `set preprocessing reduce`
- ◆ Concert Technology Library, the parameter: `IloCplex::Reduce`
- ◆ Callable Library, the parameter: `CPX_PARAM_REDUCE`

can be used to select one of the following values:

0 = no primal and dual reductions
1 = only primal reductions
2 = only dual reductions
3 = both primal and dual reductions

If your problem includes network structures, there is a possibility that ILOG CPLEX preprocessing may eliminate those structures from your model. For that reason, you should consider turning off preprocessing before you invoke the network optimizer.

The dependency checker strengthens problem reduction by detecting redundant constraints. Such reductions are usually most effective with the primal-dual barrier optimizer.

To turn on the dependency checker to strengthen reduction:

- In the Interactive Optimizer, use the command `set preprocessing dependency 1`.
- When using the Component Libraries, set the parameter `IloCplex::DepInd` or `CPX_PARAM_DEPIND`.

To reduce memory usage, presolve may compress the arrays used for storage of the original model. This can make more memory available for the use of the optimizer that the user has called. Under default settings CPLEX automatically determines, from characteristics of the model, whether to perform this compression. You can explicitly turn this feature on or off by setting the presolve compression parameter to -1 for off, or 1 for on; the default of 0 specifies the automatic setting.

To set presolve compression:

- In the Interactive Optimizer enter the command `set preprocessing compress`.
- In the Component Libraries use the parameter `IloCplex::PreCompress` or `CPX_PARAM_PRECOMPRESS`.

In case you want to save the preprocessed version of a problem:

- In the Interactive Optimizer, use the `write` command with the `pre` file type to save a binary copy to a file.
- When using the Component Libraries, use the method `IloCplex::exportModel()` or the routine `CPXwriteprob()`.

Preprocessing: Explicitly Solving the Dual

In some situations, such as a model that has many more rows than columns, it may be advantageous to have ILOG CPLEX treat your model internally as the dual formulation. Then you can call any of the linear optimizers on that formulation, and again CPLEX will report results in terms of your original formulation.

To treat your model internally as the dual formulation and have ILOG CPLEX return results in terms of your original formulation:

In the Interactive Optimizer, follow these steps:

1. If you have previously turned off the presolver, turn it back on. (The default setting of the presolver is on. Dual preprocessing is ignored when the presolver is off.) Turn the presolver on with the command `set preprocessing presolve yes`.
2. Call for dual simplex preprocessing with the command `set preprocessing dual 1`.
3. Then solve with any of CPLEX's linear optimizers.

Similarly, when using the Component Libraries, in your own application, you can first turn on the preprocessing presolver, request dual preprocessing and call the default optimizer, like this:

```
cplex.setParam(IloCplex::PreInd, IloTrue);
cplex.setParam(IloCplex::PreDual, IloTrue);
cplex.solve();

CPXsetintparam(env, CPX_PARAM_PREIND, CPX_ON);
CPXsetintparam(env, CPX_PARAM_PREDUAL, CPX_ON);
CPXlpopt(env, lp);
```

The default setting of this parameter is 0, meaning "automatic". With this setting, CPLEX examines the problem and decides whether solving the primal or dual problem will be more efficient. Currently, CPLEX performs this assessment only with the barrier optimizer. For all other optimizers the default setting causes CPLEX to solve the primal problem.

It is worth emphasizing, to those versed in linear programming theory, that the decision to solve the dual formulation of your model, via this preprocessing parameter, is entirely separate from the choice of using the dual simplex method versus the primal simplex method to perform the optimization. Although these features have theoretical underpinnings in common, it is not redundant to consider (for example) solving the dual formulation of your model with the dual simplex method; this would not simply result in the same computational path as solving the primal formulation with the primal simplex method. In the case already mentioned of a model with many more rows than columns, either simplex variant may perform much better when solving the dual formulation, due to the smaller basis matrix that is maintained.

Starting from an Advanced Basis

As another performance improvement, consider starting from an advanced basis. (The primal simplex, dual simplex, and network optimizers can start optimizing from an advanced basis if one is available; the primal-dual ILOG CPLEX Barrier Optimizer does not start from an advanced basis.) If you can start from an advanced basis, then ILOG CPLEX may iterate significantly fewer times, particularly if your current problem is similar to a problem that you have solved previously. Even when problems are different, starting from an advanced basis may still help performance. For example, if your current problem is composed of several smaller problems, an optimal basis from one of the component problems may significantly speed up solution of the larger problem.

Note that if you are solving a sequence of LP models all within one run, by entering a model, solving it, modifying the model, and solving it again, under default settings the advanced basis will be used for the last of these steps automatically.

In other cases, you can communicate the final basis from one run to the start of the next by first saving the basis to a file before the end of the first run.

To save the basis of the optimized problem to a file:

- When using the Component Libraries to optimize your problem, save by means of the method `cplex.exportModel()` or the routine `CPXwriteprob()`.
- In the Interactive Optimizer, use the `write` command with the file type `sav`.

To read an advanced basis from a saved file into the Interactive Optimizer, follow these steps:

1. Assure that the advanced start parameter is set to its default value of `yes`:
   `set advance y`.
2. Read the file with the `read` command, and specify the file type as `bas`.

Similarly, when using the Component Libraries, set the parameter `IloCplex::AdvInd` or `CPX_PARAM_ADVIND`, to `1` and call the method `Ilocplex.importModel()` or the routine `CPXreadcopybase()`.

Adjusting Parameters

After you have chosen the right optimizer and, if appropriate, you have started from an advanced basis, you may want to experiment with different parameter settings to improve performance. This section describes parameters that are most likely to affect performance of linear optimizers. (*Managing Parameters from the Callable Library* on page 70, discusses parameter settings in general.)

To adjust parameters:

- In the Interactive Optimizer, use the `set` command.
- For the Concert Technology Library, use the method `cplex.setParam()`.
- For the Callable Library, the routine `CPXsetintparam()` adjusts integer-valued parameters, and the routine `CPXsetdblparam()` adjusts parameters that take values of type `double`.

For more performance tuning suggestions, refer to the following sections:

- Memory Management and Problem Growth
- Pricing Algorithm and Gradient Parameters
- Scaling
- Refactoring Frequency
- Crash

If you find better parameter settings for your problem, save them in a parameter specification file, as explained in *Saving a Parameter Specification File* on page 339.

Memory Management and Problem Growth

As it works, ILOG CPLEX automatically handles memory allocations to accommodate the increasing size of a problem object as you modify the object through calls to modification routines in the Callable Library. The sequence of Callable Library routines that you invoke can influence the efficiency of memory management. As we show in Table 2.2 on page 62, you can control how ILOG CPLEX allocates memory through its growth parameters.

How you should set these growth parameters depends on how you build the problem object in a particular application. CPLEX will automatically manage (via a cache) most changes to prevent inefficiency when the changes will require memory re-allocations. If an application builds a large problem in small increments, you still may be able to improve performance by increasing the growth parameters. In particular, if you know reasonably accurate upper bounds on the number of rows, columns, and nonzeros, and you are building a large problem in very small pieces with many calls to the problem modification routines, then setting the growth parameters to the known upper bounds will make ILOG CPLEX perform the fewest allocations and may thus improve performance of the problem modification routines. However, overly generous upper bounds may result in excessive memory use.

Pricing Algorithm and Gradient Parameters

The gradient parameters in Table 4.2 determine the pricing algorithms that ILOG CPLEX uses. Consequently, these are the algorithmic parameters most likely to affect simplex linear programming performance. The default setting of these gradient parameters choose the pricing algorithms that are best for most problems. Moreover, the enhancements of the linear algebra routines that the ILOG CPLEX Simplex Optimizers use affect the various gradient options differently. When you are selecting alternate pricing algorithms, look at these values as guides:

- overall solution time;
- number of Phase I iterations (that is, iterations before ILOG CPLEX arrives at an initial feasible solution);
- total number of iterations.

ILOG CPLEX records those values in the log file as it works. (By default, ILOG CPLEX creates the log file in the directory where it is executing, and it names the log file `cplex.log`. *Managing Log Files: the Log File Parameter* on page 269 tells you how to rename and relocate this log file.)

If the number of iterations required to solve your problem is approximately the same as the number of rows, then you are doing well. If the number of iterations is three times greater than the number of rows (or more), then it may very well be possible to improve performance by changing the gradient parameter that determines the pricing algorithm.

Table 4.2 *Gradient Parameters*

Parameter	**Primal Simplex**	**Dual Simplex**
In Interactive Optimizer	`simplex pgradient`	`simplex dgradient`
In Concert Technology Library	`IloCplex::PPriInd`	`IloCplex::DPriInd`
In Callable Library	`CPX_PARAM_PPRIIND`	`CPX_PARAM_DPRIIND`

Table 4.3 lists acceptable values for the primal simplex pricing parameter. Table 4.4 lists values for dual simplex pricing parameter. The following paragraphs explain those values and offer advice about them.

Table 4.3 *Primal Simplex Pricing Algorithm Values*

Symbolic constant value	**Integer value**	**Pricing algorithm**
`CPX_PPRIIND_PARTIAL`	`-1`	reduced-cost pricing
`CPX_PPRIIND_AUTO`	`0 (default)`	hybrid reduced-cost and devex pricing
`CPX_PPRIIND_DEVEX`	1	devex pricing
`CPX_PPRIIND_STEEP`	2	steepest-edge pricing
`CPX_PPRIIND_STEEPQSTART`	3	steepest-edge pricing with initial slack norms
`CPX_PPRIIND_FULL`	4	full pricing

For the primal simplex pricing parameter, *reduced-cost pricing* (`-1` or `CPX_PPRIIND_PARTIAL`) is less computationally expensive, so you may prefer it for small or relatively easy problems. Try reduced-cost pricing, and watch for faster solution times. Also if your problem is dense (say, 20-30 nonzeros per column), reduced-cost pricing may be advantageous.

In contrast, if you have a more difficult problem taking many iterations to complete Phase I and arrive at an initial solution, then you should consider *devex pricing* (`1` or `CPX_PPRIIND_DEVEX`). Devex pricing benefits more from ILOG CPLEX linear algebra enhancements than does partial pricing, so it may be an attractive alternative to partial pricing in some problems. Do *not* use devex pricing, however, if your problem has many columns and relatively few rows. In such a case, the number of calculations required per iteration will usually be disadvantageous.

If you observe that devex pricing helps, then you might also consider steepest-edge pricing (`2` or `CPX_PPRIIND_STEEP`). Steepest-edge pricing is computationally more expensive than reduced-cost pricing, but it may produce the best results on difficult problems. One way of reducing the computational intensity of steepest-edge pricing is to choose steepest-edge pricing with initial slack norms (`3` or `CPX_PPRIIND_STEEPQSTART`).

For the dual simplex pricing parameter, the default value selects steepest-edge pricing with unit initial norms. That is, the default (`0` or `CPX_DPRIIND_AUTO`) automatically selects `4` or `CPX_DPRIIND_STEEPQSTART`. You may also consider starting with exact norms, since ILOG CPLEX has reduced the cost of initializing norms.

Table 4.4 *Dual Simplex Pricing Algorithm Values*

Symbolic Constant Values	Integer Value	Pricing Algorithm
`CPX_DPRIIND_AUTO`	`0` (default)	ILOG CPLEX determines automatically
`CPX_DPRIIND_FULL`	`1`	standard dual pricing
`CPX_DPRIIND_STEEP`	`2`	steepest-edge pricing
`CPX_DPRIIND_FULL_STEEP`	`3`	steepest-edge pricing in slack space
`CPX_DPRIIND_STEEPQSTART`	`4`	steepest-edge pricing with unit initial norms

Scaling

Poorly conditioned problems (that is, problems in which even minor changes in data result in major changes in solutions) may benefit from an alternative *scaling* method. Scaling attempts to rectify poorly conditioned problems by multiplying rows or columns by constants without changing the fundamental sense of the problem. If you observe that your problem has difficulty staying feasible during its solution, then you should consider an alternative scaling method.

To set an alternative scaling method:

◆ In the Interactive Optimizer, use the command `set read scale i`, substituting *0* (zero) for `i` to achieve equilibration scaling or *1* (one) for *`i`* to achieve more aggressive scaling. In certain cases, it may be advantageous to turn off scaling. To do so in the Interactive Optimizer, use the command `set read scale -1`.

◆ When using the Component Libraries, set the parameter `IloCplex::ScaInd` or `CPX_PARAM_SCAIND`. to the appropriate value.

Refactoring Frequency

ILOG CPLEX dynamically determines the frequency at which the basis of a problem is refactored in order to maximize iteration speed. On very large problems, ILOG CPLEX refactors the basis matrix infrequently. Very rarely should you consider increasing the number of iterations between refactoring. In such cases:

- In the Interactive Optimizer, use the command `set simplex refactor i` (substituting a positive integer for `i`) to change the refactoring frequency.
- When using the Component Libraries, set the parameter `IloCplex::ReInv` or `CPX_PARAM_REINV`.

Crash

It is possible to control the way ILOG CPLEX builds an initial basis through the crash parameter.

In the primal simplex optimizer, the crash setting determines how ILOG CPLEX uses the coefficients of the objective function to select the starting basis. If its value is `1` (one), ILOG CPLEX uses the coefficients to guide its selection; if its value is `0` (zero), ILOG CPLEX ignores the coefficients; if its value is `-1`, ILOG CPLEX does the opposite of what the coefficients normally suggest. These values are summarized in Table 4.5.

Table 4.5 *Values of the ILOG CPLEX Crash Parameter for the Primal Simplex Optimizer*

Value	Meaning for Primal Simplex Optimizer
`1`	Use coefficients of objective function to select basis
`0`	Ignore coefficients of objective function
`-1`	Select basis contrary to one indicated by coefficients of objective function

In the dual simplex optimizer, the crash setting determines whether ILOG CPLEX aggressively uses primal variables instead of slack variables while it still tries to preserve as much dual feasibility as possible. If its value is `1` (one), it indicates the default starting basis; if its value is `0` (zero) or `-1`, it indicates an aggressive starting basis. These values are summarized in Table 4.6.

Table 4.6 *Values of the ILOG CPLEX Crash Parameter for the Dual Simplex Optimizer*

Value	Meaning for Dual Simplex Optimizer
`1`	Use default starting basis
`0`	Use an aggressive starting basis
`-1`	Use an aggressive starting basis

To control the way ILOG CPLEX builds an initial basis:

- In the Interactive Optimizer, use the command `set simplex crash i` (substituting 1, 0, or -1 for `i`) .
- When using the Component Libraries, set the parameter `IloCplex::CraInd` or `CPX_PARAM_CRAIND`.

Diagnosing Performance Problems

While some linear programming models offer opportunities for performance tuning, others, unfortunately, entail outright performance problems that require diagnosis and correction. This section indicates how to diagnose and correct such performance problems as lack of memory or numerical difficulties.

Lack of Memory

To sustain computational speed, ILOG CPLEX tries to use only available physical memory, rather than virtual memory or paged memory. Even if your problem data fit in memory, ILOG CPLEX will need still more memory to optimize the problem. When ILOG CPLEX recognizes that only limited memory is available, it automatically makes algorithmic adjustments to compensate. These adjustments almost always reduce optimization speed. If you detect when these automatic adjustments occur, then you can determine when you need to add additional memory to your computer to sustain computational speed for your particular problem. The following sections offer guidance for you to detect these automatic adjustments.

Warning Messages

In certain cases, ILOG CPLEX issues a warning message when it cannot perform an operation, but it continues to work on the problem. Other ILOG CPLEX messages indicate that ILOG CPLEX is compressing the problem to conserve memory. These warnings mean that ILOG CPLEX finds insufficient memory available, so it is following an alternate—less desirable—path to a solution. If you provide more memory, ILOG CPLEX will return to the best path toward a solution.

Paging Virtual Memory

On systems such as UNIX, where there is virtual memory management, if you observe paging of memory to disk, then your application is incurring a performance penalty. If you increase available memory in such a case, performance will speed up dramatically.

Refactoring Frequency and Memory Requirements

The ILOG CPLEX Primal and Dual Simplex Optimizers refactor the problem basis at a rate determined by the refactor parameter:

- `simplex refactor` in the Interactive Optimizer,
- `IloCplex::ReInv` in the Concert Technology Library, and
- `CPX_PARAM_REINV` in the Callable Library.

The longer ILOG CPLEX works between refactoring, the greater the amount of memory it needs for each iteration. Consequently, one way of conserving memory is to increase the refactoring frequency by decreasing the interval between refactorings. In fact, if little memory is available to it, ILOG CPLEX will automatically decrease the refactoring interval in order to use less memory at each iteration.

Since refactoring is an expensive operation, decreasing the refactoring interval will generally slow performance. You can tell whether performance is being degraded in this way by checking the iteration log file.

In an extreme case, lack of memory may force ILOG CPLEX to refactor at every iteration, and the impact on performance will be dramatic. If you provide more memory in such a situation, the benefit will be tremendous.

Preprocessing and Memory Requirements

By default, the ILOG CPLEX presolver and aggregator are active. That is, ILOG CPLEX automatically preprocesses your problem before optimizing, and this preprocessing requires memory. If memory is extremely tight, consider turning off preprocessing.

To turn off preprocessing:

- ◆ In the Interactive Optimizer, use the command `set preprocessing presolve 0`.
- ◆ When using the Component Libraries, set the parameter `IloCplex::PreInd` or `CPX_PARAM_PREIND`.

Numerical Difficulties

ILOG CPLEX is designed to handle the numerical difficulties of linear programming automatically. In this context, numerical difficulties mean such phenomena as repeated occurrence of singularities, little or no progress toward realizing the objective function value, little or no progress in scaled infeasibility, repeated problem perturbations, repeated occurrences of the problem becoming infeasible. While ILOG CPLEX will usually achieve an optimal solution in spite of these difficulties, you can help it do so more efficiently. This section describes situations in which you can help.

Some problems will not be solvable even after you take the measures we suggest. Such problems are so poorly conditioned that their optimal solutions are beyond the numerical precision of your computer.

Numerically Sensitive Data

There is no absolute link between the form of data in a model and the numerical difficulty the problem poses. Nevertheless, certain choices in how you present the data to CPLEX can have an adverse effect.

Placing large upper bounds (say, in the neighborhood of $1e^9$ to $1e^{12}$) on individual variables can cause difficulty during Presolve. If you intend for such large bounds to mean “no bound is really in effect” it is better to simply not include such bounds in the first place.

Large coefficients anywhere in the model can likewise cause trouble at various points in the solution process. Even if the coefficients are of more modest size, a wide variation (say, six or more orders of magnitude) in coefficients found in the objective function or right hand side, or in any given row or column of the matrix, can cause difficulty either in Presolve when it makes substitutions, or in the optimizer routines, particularly the barrier optimizer, as convergence is approached.

A related source of difficulty is the form of rounding when fractions are represented as decimals; expressing 1/3 as .33333333 on a computer that in principle computes values to about 16 digits can introduce an apparent "exact" value that will be treated as given but may not represent what you intend. This difficulty is compounded if similar or related values are represented a little differently elsewhere in the model. The underlying principle behind all these cautions is that "information" contained down in the 8th or 10th decimal place of data needs to convey actual meaning or the optimizer may start to draw false conclusions.

Measuring Problem Sensitivity with Basis Condition Number

Ill-conditioned matrices are sensitive to minute changes in problem data. That is, in such problems, small changes in data can lead to very large changes in the reported problem solution. ILOG CPLEX provides a *basis condition number* to measure the sensitivity of a linear system to the problem data. You might also think of the basis condition number as the number of places in precision that can be lost.

For example, if the basis condition number at optimality is $1e^{13}$, then a change in a single matrix coefficient in the thirteenth place may dramatically alter the solution. Furthermore, since many computers provide about 16 places of accuracy in double precision, only three accurate places are left in such a solution. Even if an answer is obtained, perhaps only the first three significant digits are reliable.

Because of this effective loss of precision for matrices with high basis condition numbers, ILOG CPLEX may be unable to select an optimal basis. In other words, a high basis condition number can make it impossible to find a solution.

- In the Interactive Optimizer, use the command `display solution kappa` in order to see the basis condition number of a resident basis matrix.
- In the Concert Technology Library, use the method `cplex.getQuality(IloCplex::Kappa)`.
- In the Callable Library, use the routine `CPXgetdblquality()` to access the condition number in the double-precision variable `dvalue`, like this:

```
status = CPXgetdblquality(env, lp, &dvalue, CPX_KAPPA);
```

Repeated Singularities

Whenever ILOG CPLEX encounters a singularity, it removes a column from the current basis and proceeds with its work. ILOG CPLEX reports such actions to the log file (by default) and to the screen (if you are working in the Interactive Optimizer or if the message-to-screen indicator `CPX_PARAM_SCRIND` is set to `1` (one)). Once it finds an optimal solution under these conditions, ILOG CPLEX will then re-include the discarded column to be sure that no better solution is available. If no better objective value can be obtained, then the problem has been solved. Otherwise, ILOG CPLEX continues its work until it reaches optimality.

Occasionally, new singularities occur, or the same singularities recur. ILOG CPLEX observes a limit on the number of singularities it tolerates. By default, the limit is ten. After encountering ten singularities, ILOG CPLEX will save in memory the best factorable basis found so far and stop its solution process. You may want to save this basis to a file for later use.

To change the number of singularities that ILOG CPLEX tolerates:

- ◆ In the Interactive Optimizer, use the command `set simplex limits singularity` *`i`*, substituting a non-negative value for *`i`*.
- ◆ When using the Component Libraries, set the parameter `IloCplex::SingLim` or `CPX_PARAM_SINGLIM`.

To save the best factorable basis found so far in the Interactive Optimizer, use the `write` command with the file type `sav`. When using the Component Libraries, use the method `cplex.exportModel()` or the routine `CPXwriteprob()`.

If ILOG CPLEX encounters repeated singularities in your problem, you may want to try alternative scaling on the problem (rather than simply increasing ILOG CPLEX tolerance for singularities). *Scaling* on page 105 explains how to try alternative scaling.

If alternate scaling does not help, another tactic to try is to increase the Markowitz tolerance. The Markowitz tolerance controls the kinds of pivots permitted. If you set it near its maximum value of `0.99999`, it may make iterations slower but more numerically stable. *Inability to Stay Feasible* on page 111 shows how to change the Markowitz tolerance.

If none of these ideas help, you may need to alter the model of your problem. Consider removing the offending variables manually from your model, and review the model to find other ways to represent the functions of those variables.

Stalling Due to Degeneracy

Highly degenerate linear programs tend to stall optimization progress in the primal and dual simplex optimizers. When stalling occurs with the primal simplex optimizer, ILOG CPLEX automatically perturbs the *variable bounds*; when stalling occurs with the dual simplex optimizer, ILOG CPLEX perturbs the *objective function*.

In either case, perturbation creates a different but closely related problem. Once ILOG CPLEX has solved the perturbed problem, it removes the perturbation by resetting problem data to their original values.

If ILOG CPLEX automatically perturbs your problem early in the solution process, you should consider starting the solution process yourself with a perturbation. (Starting in this way will save the time that would be wasted if you first allowed optimization to stall and then let ILOG CPLEX perturb the problem automatically.)

To start perturbation yourself:

- In the Interactive Optimizer, use the command `set simplex perturbation y i` where the first option, `y`, indicates yes and the second option, `i`, (which you fill in with a value appropriate for your problem) indicates how much perturbation to introduce.
- When using the Component Libraries set the parameter `IloCplex::PerInd` or `CPX_PARAM_PERIND` to turn on perturbation from the start, and set the parameter `IloCplex::EpPer` or `CPX_PARAM_EPPER` to any positive value greater than $1e^{-8}$.

If you observe that your problem has been perturbed more than once, then consider whether the simplex perturbation-limit parameter is too large. The perturbation-limit parameter determines the number of iterations ILOG CPLEX tries before it assumes the problem has stalled. At its default value, `0` (zero), ILOG CPLEX determines internally how many iterations to perform before declaring a stall. If you set this parameter to some other nonnegative integer, then ILOG CPLEX uses that limit to determine when a problem has stalled. If you reduce the perturbation-limit parameter, you may be able to reduce the number of times the problem is necessarily perturbed.

To reduce the simplex perturbation-limit parameter:

- In the Interactive Optimizer, use the command
`set simplex limits perturbation i`, substituting a smaller value for `i`.
- When using the Component Libraries, set the parameter `IloCplex::PerLim` or `CPX_PARAM_PERLIM`.

Inability to Stay Feasible

If a problem repeatedly becomes infeasible in Phase II (that is, after ILOG CPLEX has achieved a feasible solution), then numerical difficulties may be occurring. It may help to increase the Markowitz tolerance in such a case. By default, its value is `0.01`, and suitable values range from `0.0001` to `0.99999`.

To increase Markowitz tolerance:

- In the Interactive Optimizer, use the command
`set simplex tolerances markowitz n`, substituting a greater value for `n`.
- When using the Component Libraries set the parameter `IloCplex::EpMrk` or `CPX_PARAM_EPMRK`.

Sometimes slow progress in Phase I (the period when ILOG CPLEX calculates the first feasible solution) is due to similar numerical difficulties, less obvious because feasibility is not gained and lost. In the progress reported in the log file, an increase in the printed sum of infeasibilities may be a symptom of this case. If so, it may be worth while to set a higher Markowitz tolerance, just as in the more obvious case of numerical difficulties in Phase II.

Diagnosing LP Infeasibility

ILOG CPLEX reports statistics about any problem that it optimizes. For infeasible solutions, it reports values that you can analyze to determine where your problem formulation proved infeasible. In certain situations, you can then alter your problem formulation or change ILOG CPLEX parameters to achieve a satisfactory solution. This section explains how to analyze such reports and indicates steps to take to alter your problem formulation or to change ILOG CPLEX parameters.

- When the ILOG CPLEX *Primal* Simplex Optimizer terminates with an infeasible basic solution, it calculates dual variables and reduced costs relative to the Phase I objective; that is, relative to the infeasibility function. The Phase I objective function depends on the current basis. Consequently, if you use the primal simplex optimizer with various parameter settings, an infeasible problem will produce different objective values and different solutions.
- When the CPLEX *Dual* Simplex Optimizer terminates and reports an unbounded solution, then the original problem is infeasible. When the dual simplex optimizer terminates and reports an infeasible problem, the original problem is either infeasible, too, or unbounded.

Table 4.7 summarizes these implications.

Table 4.7 *Implications of Dual Solutions for Primal Formulations*

If the dual is	Then the primal is
unbounded	infeasible
infeasible	either infeasible or unbounded

The Effect of Preprocessing on Feasibility

CPLEX preprocessing may declare a model infeasible before the selected optimization algorithm begins. This saves considerable execution time in most cases. It is important, when this is the outcome, to understand that there are two classes of reductions performed by the preprocessor.

Reductions that are independent of the objective function are called primal reductions; those that are independent of the right-hand side are called dual reductions. Preprocessing operates on the assumption that the model being solved is expected by the user to be feasible and that a finite optimal solution exists. If this assumption is false, then the model is either infeasible or no bounded optimal solutions exist, i.e. unbounded. Since primal reductions are independent of the objective function, they cannot detect unboundedness, they can only detect infeasibility. Similarly, dual reductions can only detect unboundedness.

Thus, to aid analysis of an infeasible or unbounded declaration by the preprocessor, a parameter is provided that the user can set, so that the optimization can be rerun to assure that the results reported by the preprocessor can be interpreted. If a model is declared by the preprocessor to be infeasible or unbounded and the user believes that it might be infeasible, the parameter `IloCplex::Reduce` or `CPX_PARAM_REDUCE` (`set preprocessing reduce` in the Interactive Optimizer) can be set to `1` by the user, and the preprocessor will only perform primal reductions. If the preprocessor still finds inconsistency in the model, it will be declared by the preprocessor to be infeasible, instead of infeasible or unbounded. Similarly, setting the parameter to `2` means that if the preprocessor detects unboundedness in the model, it will be declared unambiguously to be unbounded.

These parameters are intended for diagnostic use, as turning off reductions will usually have a negative impact on performance of the optimization algorithms in the normal (feasible and bounded) case.

Coping with an Ill-Conditioned Problem or Handling Unscaled Infeasibilities

By default, ILOG CPLEX *scales* a problem before attempting to solve it. After it finds an optimal solution, it then checks for any violations of optimality or feasibility in the original, *unscaled* problem. If there is a violation of reduced cost (indicating nonoptimality) or of a bound (indicating infeasibility), ILOG CPLEX reports both the maximum scaled and unscaled feasibility violations.

Unscaled infeasibilities are rare, but they may occur when a problem is ill-conditioned. For example, a problem containing a row in which the coefficients have vastly different magnitude is ill-conditioned in this sense and may result in unscaled feasibilities.

It may be possible to produce a better solution anyway in spite of unscaled infeasibilities, or it may be necessary for you to revise the coefficients. To determine which way to go, we recommend these steps in such a case:

1. Use the command `display solution quality` in the Interactive Optimizer to locate the infeasibilities.
2. Examine the coefficient matrix for poorly scaled rows and columns.
3. Evaluate whether you can change unnecessarily large or small coefficients.
4. Consider alternate scalings.

You may also be able to re-optimize the problem successfully after you reduce optimality tolerance, as explained in *Maximum Reduced-Cost Infeasibility* on page 114, or after you reduce feasibility tolerance, as explained in *Maximum Bound Infeasibility: Identifying Largest Bound Violation* on page 114. When you change these tolerances, ILOG CPLEX may produce a better solution to your problem, but lowering these tolerances sometimes produces erratic behavior or an unstable optimal basis.

Check the basis condition number, as explained in *Measuring Problem Sensitivity with Basis Condition Number* on page 109. If the condition number is fairly low (for example, as little as *1e*^{5} or less), then you can be confident about the solution. If the condition number is high, or if reducing tolerance does not help, then you must revise the problem model because the current model may be too ill-conditioned to produce a numerically reliable result.

Interpreting Solution Statistics

By default, individual infeasibilities are written to a log file but not displayed on the screen. To display the infeasibilities on your screen, use the command
`set output logonly y cplex.log`.

Regardless of whether a solution is infeasible or optimal, the command
`display solution quality` in the Interactive Optimizer displays the bound and reduced-cost infeasibilities for both the scaled and unscaled problem. In fact, it displays the following summary statistics for both the scaled and unscaled problem:

- maximum bound infeasibility, that is, the largest bound violation;
- maximum reduced-cost infeasibility;
- maximum row residual;
- maximum dual residual;
- maximum absolute value of a variable, a slack variable, a dual variable, and a reduced cost.

The following sections discuss these summary statistics in greater detail.

Maximum Bound Infeasibility: Identifying Largest Bound Violation

The maximum bound infeasibility identifies the largest bound violation. This information may help you determine the cause of infeasibility in your problem. If the largest bound violation exceeds the feasibility tolerance of your problem by only a small amount, then you may be able to get a feasible solution to the problem by increasing the feasibility tolerance.

- To increase the feasibility tolerance of your problem in the Interactive Optimizer, use the command `set simplex tolerances feasibility n`, substituting a smaller value for *n*. Its range is between *1e*$^{-9}$ and *0.1*. Its default value is *1e*$^{-06}$.
- To change the infeasibility tolerance when using the Component Libraries set the parameter `IloCplex::EpRHS` or `CPX_PARAM_EPRHS`.

Maximum Reduced-Cost Infeasibility

The maximum reduced-cost infeasibility identifies a value for the optimality tolerance that would cause ILOG CPLEX to perform additional iterations. It refers to the infeasibility in the dual slack associated with reduced costs. Whether ILOG CPLEX terminated with an optimal or infeasible solution, if the maximum reduced-cost infeasibility is only slightly smaller in absolute value than the optimality tolerance, then solving the problem with a smaller optimality tolerance may result in an improvement in the objective function.

To lower the optimality tolerance in your problem in the Interactive Optimizer, use the command `set simplex tolerances optimality n`, substituting a lower value for `n`. Its range is between *1e*$^{-9}$ and *0.1*. Its default value is *1e*$^{-06}$. To lower the optimality tolerance when using the Component Libraries set the parameter `IloCplex::EpOpt` or `CPX_PARAM_EPOPT`.

Maximum Row Residual

The maximum row residual identifies the maximum constraint violation. ILOG CPLEX Simplex Optimizers control these residuals only indirectly by applying numerically sound methods to solve the given linear system. When ILOG CPLEX terminates with an infeasible solution, all infeasibilities will appear as bound violations on structural or slack variables, not constraint violations. The maximum row residual may help you determine whether a model of your problem is poorly scaled, or whether the final basis (whether it is optimal or infeasible) is ill-conditioned.

Maximum Dual Residual

The maximum dual residual indicates whether the current optimality tolerance is set appropriately. If the maximum dual residual exceeds the optimality tolerance, ILOG CPLEX may stall before it reaches an optimal solution. In particular, if ILOG CPLEX stalls during Phase I after almost reducing the sum of infeasibilities to 0 (zero), then you may be able to find a feasible solution if you increase the optimality tolerance.

To increase the optimality tolerance in your problem in the Interactive Optimizer, use the command `set simplex tolerances optimality n`, substituting a larger value for `n`. Its range is between *1e*$^{-09}$ and *0.1*. Its default value is *1e*$^{-06}$. To increase the optimality tolerance when using the Component Libraries set the parameter `IloCplex::EpOpt` or `CPX_PARAM_EPOPT`.

Maximum Absolute Values: Detecting Ill-Conditioned Problems

When you are trying to determine whether your problem is ill-conditioned, you need to consider the following maximum absolute values, all available in the infeasibility analysis that ILOG CPLEX provides you:

- variables;
- slack variables;
- dual variables;
- reduced costs (that is, dual slack variables).

When using the Component Libraries, use the method `cplex.getQuality()` or the routine `CPXgetdblquality()` to access the information provided by the command `display solution quality` in the Interactive Optimizer.

If you determine from this analysis that your model is indeed ill-conditioned, then you need to reformulate it. *Coping with an Ill-Conditioned Problem or Handling Unscaled Infeasibilities* on page 113 outlines steps to follow in this situation.

Finding a Set of Irreducibly Inconsistent Constraints

If ILOG CPLEX reports that your problem is infeasible, then you should invoke the ILOG CPLEX infeasibility finder to save time and effort in the development process. This diagnostic tool computes a set of infeasible constraints and column bounds that would be feasible if one of them (a constraint or variable) were removed. Such a set is known as *irreducibly inconsistent*.

To work, the infeasibility finder must have a problem that satisfies two conditions:

- the problem has been optimized using the primal method or barrier with crossover, and
- the problem has terminated with an infeasible basic solution to the primal problem.

When the ILOG CPLEX presolver detects infeasibility during preprocessing, no optimization has yet taken place. Furthermore, since the presolver may perform many passes on a problem, the reason that it identifies a row as infeasible may not be obvious. To run the infeasibility finder and to see solution statistics in such a case, you should first turn off ILOG CPLEX preprocessing before you optimize, as explained in *Preprocessing: Presolver and Aggregator* on page 98, before you invoke the infeasibility finder.

Also if you are licensed to use the primal-dual ILOG CPLEX Barrier Optimizer, remember that you may call it optionally without simplex crossover. In such a case, ILOG CPLEX will not produce the infeasible basis that the infeasibility finder needs, so if you want to run the infeasibility finder with the primal-dual barrier optimizer, then you must call that optimizer with simplex crossover turned on.

Infeasibility Finder in the Interactive Optimizer

To invoke the infeasibility finder and to display part of its findings in the Interactive Optimizer, use the command `display iis`. By default, ILOG CPLEX records all its findings in a log file. To send these findings to your screen as well, use the command `set output logonly y cplex.log`.

You can also write an IIS file from the Interactive Optimizer and then examine it with your preferred text editor to see all the constraints and bounds in the irreducibly inconsistent set.

For an example of how to use the infeasibility finder and how to interpret its results, see *Example: Output from the Infeasibility Finder in the Interactive Optimizer* on page 117.

Infeasibility Finder in the Component Libraries

When using the Component Libraries, to specify the infeasibility finder, set the parameter `IloCplex::IISInd` or `CPX_PARAM_IISIND`. Its default value of `0` (zero) invokes an algorithm that requires minimal computation time but it may generate a large set of inconsistent constraints. Its alternative value of `1` (one) may take longer but generates a minimal set of irreducibly inconsistent constraints. After you have specified the kind of IIS to generate, use the method `cplex.getIIS()` or the routine `CPXfindiis()` to tell ILOG CPLEX to compute the set. Then use the method `cplex.out()` or the routines `CPXdisplayiis()` or `CPXiiswrite()` to output the results for review.

Correcting Multiple Infeasibilities

The infeasibility finder will find only one irreducibly inconsistent set (IIS), though a given problem may contain many independent IISs. Consequently, even after you detect and correct one such IIS in your problem, it may still remain infeasible. In such a case, you need to run the infeasibility finder more than once to detect those multiple causes of infeasibility in your problem.

Example: Output from the Infeasibility Finder in the Interactive Optimizer

After you have optimized a problem and CPLEX has terminated with a primal infeasible basic solution, then you can invoke the CPLEX infeasibility finder on this optimized problem and its infeasible basic solution. The ILOG CPLEX infeasibility finder will compute an *irreducibly inconsistent set* (IIS) of constraints and column bounds from your problem and record this IIS in a log file along with other useful information to help you locate the source of infeasibility and aid you in revising or reformulating your problem model.

If you want ILOG CPLEX to display this additional information on your screen in the Interactive Optimizer, use the command `set output logonly yes`. After that command, invoke the infeasibility finder with the command `display iis. ILOG CPLEX` will respond like this:

```
Starting Infeasibility Finder Algorithm...
Performing row sensitivity filter
Performing column sensitivity filter

Number of rows in the iis: 3
Number of columns in the iis: 3
Names of rows in the iis:
NODE5    (fixed)
D7       (lower bound)
D8       (lower bound)
Names of columns in the iis:
T25      (upper bound)
T35      (upper bound)
T47      (upper bound)
Iis Computation Time =    0.01 sec.
```

As you can see, ILOG CPLEX states how many rows and columns comprise the IIS. It also tells the row and column names, and it identifies the bound causing the infeasibility. In this example, all the columns in the IIS are limited by their upper bound. If you remove any of the upper bounds on those columns, then the IIS becomes feasible. The bound information about rows is really needed only for *ranged* rows. In the case of ranged rows, the bound indicates whether the row lies at the lower or upper end of the range of right-hand side values. For other kinds of rows, there is only one possible right-hand side value at which the row can lie. Greater-than constraints must lie at their lower bound. Less-than constraints must lie at their upper bound. Equality constraints are fixed.

Example: Writing an IIS-Type File

After you have invoked the infeasibility finder with the `display iis` command, if you want additional information to determine the cause of infeasibility, use the `write` command and the file type `iis` to generate a ILOG CPLEX LP format file containing each individual constraint and bound in the IIS. You can then use the `xecute` command to call an ordinary text editor during your ILOG CPLEX session to examine the contents of that IIS file. It will look something like this example:

```
CPLEX> write infeas.iis

Starting Infeasibility Finder Algorithm...
Performing row sensitivity filter
Performing column sensitivity filter
Irreducibly inconsistent set written to file ‘infeas.iis’.

CPLEX> xecute edit infeas.iis

Minimize
subject to
\Rows in the iis:
 NODE5: T25 + T35 - T57 - T58  = 0
 D7:    T47 + T57 >= 20
 D8:    T58 >= 30
\Columns in the iis:
Bounds
 T25 <= 10
 T35 <= 10
 T47 <= 2
\Non-iis columns intersecting iis rows:
 T57 Free
 T58 Free
```

In this example, you see that the bounds on *T25* and *T35* combine with the row *NODE5* to imply that $T57 + T58 \leq 20$. However, row *D7* and the bound on *T47* imply that $T57 \geq 18$. Since row *D8* requires $T58 \geq 30$, we see that $T57 + T58 \geq 48$, so the constraints and bounds are infeasible. Notice that every constraint and bound contributes to this infeasibility, according to the definition of an IIS. There are, in consequence, many different ways to modify such a problem to make it feasible. The “right” change will depend on your knowledge of the problem.

When ILOG CPLEX records the constraints and bounds of an IIS, it also lists as *free* all columns that intersect one or more rows in the IIS but do not have bounds in the IIS. This portion of the file ensures that when you read the file into ILOG CPLEX, the resulting problem satisfies the definition of an IIS. After you read in such a file, you can perform additional problem analysis within your ILOG CPLEX session.

Example: Interpreting a Cumulative Constraint

In the example that we have been looking at, there were sufficiently few rows and column bounds in the IIS for us to see the cause of infeasibility at a glance. In contrast, other IIS files may contain so many rows and columns that it becomes difficult to see the cause of infeasibility. When an IIS contains many equality constraints and only a few bounds, for example, this phenomenon commonly occurs. In such a situation, the equality constraints transfer the infeasibility from one part of the model to another until eventually no more transfers can occur. Consequently, such an IIS file will also contain a cumulative constraint consisting of the sum of all the equality rows. This cumulative constraint can direct you toward the cause of infeasibility, as the following sample IIS illustrates:

```
Minimize
subject to
\Rows in the iis:
 2:   - x24 + x97 + x98 - x99 - x104  = -7758
 3:   - x97 + x99 + x100 - x114 - x115  = 0
 4:   - x98 + x104  = 0
 10:  - x105 + x107 + x108 - x109  = -151
 11:  - x108 + x109 + x110 - x111  = -642
 12:  - x101 - x102 - x110 + x111 + x112 + x113 - x117  = -2517
 13:  - x112 + x117 + x118 - x119  = -186
 14:  - x118 + x119 + x120 + x121 - x122 - x123  = -271
 15:  - x120 + x122  = -130
 16:  - x121 + x123 + x124 - x125  = -716
 17:  - x124 + x125 + x126 - x127  = -2627
 18:  - x126 + x127 + x128 - x129  = -1077
 19:  - x128 + x129 + x130 - x131  = -593
 249: - x100 + x101 + x103  = 0
 251: - x113 + x114 + x116  = 0
\Sum of equality rows in iis:
\  - x24 - x102 + x103 - x105 + x107 - x115 + x116 + x130 - x131 = -16668
\Columns in the iis:
Bounds
 x24 <= 14434
 x102 = 0
 x103 = 0
 x105 = 0
 x107 = 0
 x115 = 0
 x116 = 0
 x130 = 0
 x131 = 0
\Non-iis columns intersecting iis rows:
 x97 Free
 x98 Free
 x99 Free
.
.
.
End
```

Since there are 15 rows in this IIS file, the cause of infeasibility is not immediately obvious. However, if we look at the sum of the bounds on the columns, we see that

$- x24 - x102 + x103 - x105 + x107 - x115 + x116 + x130 - x131 \geq -14434$, so it is impossible to satisfy the sum of equality rows. Therefore, to correct the infeasibility, we must alter one or more of the bounds, or we must increase one or more of the right-hand side values.

Example: Setting a Time Limit on the Infeasibility Finder

The ILOG CPLEX infeasibility finder will stop when its total runtime exceeds the limit set by the command `set timelimit`. The infeasibility finder works by removing constraints and bounds that cannot be involved in the IIS, so it can provide partial information about an IIS when it reaches its time limit. The collection of constraints and bounds it offers then do not strictly satisfy the definition of an IIS. However, the collection does contain a true IIS within it, and frequently it provides enough information for you to diagnose the cause of infeasibility in your problem. When it reaches the time limit, ILOG CPLEX output indicates that it has found only a partial IIS. The following example illustrates this idea. In it, we set a time limit and then invoke the feasibility finder.

```
CPLEX> set timelimit 2
CPLEX> display iis

Starting Infeasibility Finder Algorithm...
Performing row sensitivity filter
Infeasibility Finder Algorithm exceeded time limit.
Partial infeasibility output available.

Number of rows in the (partial) iis: 101
Number of columns in the (partial) iis: 99
```

Tactics for Interpreting IIS Output

The size of the IIS reported by ILOG CPLEX depends on many factors in the problem model. If an IIS contains hundreds of rows and columns, you may find it hard to determine the cause of the infeasibility. Fortunately, there are tactics to help you interpret IIS output:

- Consider selecting an alternative IIS algorithm. The default algorithm emphasizes computation speed, and it may give rise to a relatively large IIS. If so, try setting the `iisfind` parameter to `1` (one) to invoke the alternative algorithm, and then run the infeasibility finder again. Normally, the resulting IIS is smaller because the alternative algorithm emphasizes finding a minimal IIS at the expense of computation speed.
- If the problem contains equality constraints, examine the cumulative constraint consisting of the sum of the equality rows. As we illustrated in one of the examples, the cumulative constraint can simplify your interpretation of the IIS output. More generally, if you take other linear combinations of rows in the IIS, that may also help. For example, if you add an equality row to an inequality row, the result may yield a simpler inequality row.
- Try preprocessing with the ILOG CPLEX presolver and aggregator. The presolver may even detect infeasibility by itself. If not, running the infeasibility finder on the presolved problem may help by reducing the problem size and removing extraneous constraints that do not directly cause the infeasibility but still appear in the IIS. Similarly, running

the infeasibility finder on an aggregated problem may help because the aggregator performs substitutions that may remove extraneous variables that clutter the IIS output. More generally, if you perform substitutions, you may simplify the output so that it can be interpreted more easily.

- Other simplifications of the constraints in the IIS may make it easier to interpret the IIS. We mean such simplifications as combining variables, multiplying constraints by constants, and rearranging sums.

Example: Using a Starting Basis in an LP Problem

This example shows you how to use a basis to start an optimization from an advanced point.

Example ilolpex6.cpp

The example, `ilolpex6.c`, resembles one you may have studied in the manual *Getting Started with ILOG CPLEX*, `ilolpex1.c`. This example differs from that earlier one in these ways:

- Arrays are constructed using the `populatebyrow` method, and thus no command line arguments are needed to select a construction method.
- In the `main` routine, the arrays `cstat` and `rstat` set the status of the initial basis.
- After the problem data has been copied into the problem object, the *basis* is copied by a call to `cplex.getStatuses()`.
- After the problem has been optimized, the iteration count is printed. For the given data and basis, the basis is optimal, so no iterations are required to optimize the problem.

The main program starts by declaring the environment and terminates by calling method `end()` for the environment. The code in between is encapsulated in a try block that catches all Concert Technology exceptions and prints them to the C++ error stream `cerr`. All other exceptions are caught as well, and a simple error message is issued. Next the model object and the `cplex` object are constructed. The function `populatebycolumn()` builds the problem object and, as we noted earlier, `cplex.getStatuses()` copies the advanced starting basis.

Complete Program

The complete program, `ilolpex6.cpp`, appears here or online in the standard distribution

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


static void
```

```
   populatebycolumn (IloModel model, IloNumVarArray var, IloRangeArray rng);

int
main (int argc, char **argv)
{
   IloEnv   env;
   try {
      IloModel model(env, “example”);

      IloNumVarArray var(env);
      IloRangeArray  rng(env);
      populatebycolumn (model, var, rng);

      IloCplex cplex(model);

      IloCplex::BasisStatusArray cstat(env), rstat(env);
      cstat.add(IloCplex::AtUpper);
      cstat.add(IloCplex::Basic);
      cstat.add(IloCplex::Basic);
      rstat.add(IloCplex::AtLower);
      rstat.add(IloCplex::AtLower);
      cplex.setStatuses(cstat, var, rstat, rng);
      cplex.solve();

      cplex.out() << “Solution status = “ << cplex.getStatus() << endl;
      cplex.out() << “Solution value  = “ << cplex.getObjValue() << endl;
      cplex.out() << “Iteration count = “ << cplex.getNiterations() << endl;

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      env.out() << “Values        = “ << vals << endl;
      cplex.getSlacks(vals, rng);
      env.out() << “Slacks        = “ << vals << endl;
      cplex.getDuals(vals, rng);
      env.out() << “Duals         = “ << vals << endl;
      cplex.getReducedCosts(vals, var);
      env.out() << “Reduced Costs = “ << vals << endl;

      cplex.exportModel(“lpex6.lp”);
   }
   catch (IloException& e) {
      cerr << “Concert exception caught: “ << e << endl;
   }
   catch (...) {
      cerr << “Unknown exception caught” << endl;
   }

   env.end();
   return 0;
}  // END main


static void
```

```
populatebycolumn (IloModel model, IloNumVarArray x, IloRangeArray c)
{
   IloEnv env = model.getEnv();

   IloObjective obj = IloMaximize(env);
   c.add(IloRange(env, -IloInfinity, 20.0));
   c.add(IloRange(env, -IloInfinity, 30.0));

   x.add(IloNumVar(obj(1.0) + c[0](-1.0) + c[1]( 1.0), 35.0, 40.0));
   x.add(obj(2.0) + c[0]( 1.0) + c[1](-3.0));
   x.add(obj(3.0) + c[0]( 1.0) + c[1]( 1.0));

   model.add(obj);
   model.add(c);

}  // END populatebycolumn
```

Example lpex6.c

The example, `lpex6.c`, resembles one you may have studied in the *ILOG CPLEX Getting Started* manual, `lpex1.c`. This example differs from that earlier one in these ways:

- ◆ In the `main` routine, the arrays `cstat` and `rstat` set the status of the initial basis.
- ◆ After the problem data has been copied into the problem object, the *basis* is copied by a call to `CPXcopybase()`.
- ◆ After the problem has been optimized, the iteration count is printed. For the given data and basis, the basis is optimal, so no iterations are required to optimize the problem.

The application begins with declarations of arrays to store the solution of the problem. Then, before it calls any other ILOG CPLEX routine, the application invokes the Callable Library routine `CPXopenCPLEX()` to initialize the ILOG CPLEX environment. Once the environment has been initialized, the application calls other ILOG CPLEX Callable Library routines, such as `CPXsetintparam()` with the argument `CPX_PARAM_SCRIND` to direct output to the screen and most importantly, `CPXcreateprob()` to create the problem object. The routine `populatebycolumn()` builds the problem object, and as we noted earlier, `CPXcopybase()` copies the advanced starting basis.

Before the application ends, it calls `CPXfreeprob()` to free space allocated to the problem object and `CPXcloseCPLEX()` to free the environment.

Complete Program

The complete program, `lpex6.c`, appears here or online in the standard distribution

```
#include <ilcplex/cplex.h>

/* Bring in the declarations for the string functions */
```

```
#include <string.h>

/* Include declaration for function at end of program */

#ifndef  CPX_PROTOTYPE_MIN

static int
   populatebycolumn  (CPXENVptr env, CPXLPptr lp);

#else

static int
   populatebycolumn ();

#endif


/* The problem we are optimizing will have 2 rows, 3 columns
   and 6 nonzeros.  */

#define NUMROWS    2
#define NUMCOLS    3
#define NUMNZ      6

#ifndef  CPX_PROTOTYPE_MIN
int
main (void)
#else
int
main ()
#endif
{
   char     probname[16];  /* Problem name is max 16 characters */
   int      cstat[NUMCOLS];
   int      rstat[NUMROWS];

   /* Declare and allocate space for the variables and arrays where we
      will store the optimization results including the status, objective
      value, variable values, dual values, row slacks and variable
      reduced costs. */

   int      solstat;
   double   objval;
   double   x[NUMCOLS];
   double   pi[NUMROWS];
   double   slack[NUMROWS];
   double   dj[NUMCOLS];


   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status;
   int           i, j;
   int           cur_numrows, cur_numcols;

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no output,
      so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
      be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

   if ( env == NULL ) {
      char  errmsg[1024];
      fprintf (stderr, “Could not open CPLEX environment.\n”);
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, “%s”, errmsg);
      goto TERMINATE;
   }

   /* Turn on output to the screen */

   status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
   if ( status ) {
      fprintf (stderr,
               “Failure to turn on screen indicator, error %d.\n”, status);
      goto TERMINATE;
   }

   /* Create the problem. */

   strcpy (probname, “example”);
   lp = CPXcreateprob (env, &status, probname);

   /* A returned pointer of NULL may mean that not enough memory
      was available or there was some other problem.  In the case of
      failure, an error message will have been written to the error
      channel from inside CPLEX.  In this example, the setting of
      the parameter CPX_PARAM_SCRIND causes the error message to
      appear on stdout. */

   if ( lp == NULL ) {
      fprintf (stderr, “Failed to create LP.\n”);
      goto TERMINATE;
   }

   /* Now populate the problem with the data. */

   status = populatebycolumn (env, lp);

   if ( status ) {
      fprintf (stderr, “Failed to populate problem data.\n”);
      goto TERMINATE;
   }

   /* We assume we know the optimal basis.  Variables 1 and 2 are basic,
      while variable 0 is at its upper bound */

   cstat[0] = CPX_AT_UPPER;
   cstat[1] = CPX_BASIC;
   cstat[2] = CPX_BASIC;

   /* The row statuses are all nonbasic for this problem */

   rstat[0] = CPX_AT_LOWER;
   rstat[1] = CPX_AT_LOWER;

   /* Now copy the basis */

   status = CPXcopybase (env, lp, cstat, rstat);
   if ( status ) {
      fprintf (stderr, “Failed to copy the basis.\n”);
      goto TERMINATE;
   }

   /* Optimize the problem and obtain solution. */

   status = CPXlpopt (env, lp);
   if ( status ) {
      fprintf (stderr, “Failed to optimize LP.\n”);
      goto TERMINATE;
   }

   status = CPXsolution (env, lp, &solstat, &objval, x, pi, slack, dj);
   if ( status ) {
      fprintf (stderr, “Failed to obtain solution.\n”);
      goto TERMINATE;
   }


   /* Write the output to the screen. */

   printf (“\nSolution status = %d\n”, solstat);
   printf (“Solution value  = %f\n”, objval);
   printf (“Iteration count = %d\n\n”, CPXgetitcnt (env, lp));

   /* The size of the problem should be obtained by asking CPLEX what
      the actual size is, rather than using sizes from when the problem
      was built.  cur_numrows and cur_numcols store the current number
      of rows and columns, respectively.  */

   cur_numrows = CPXgetnumrows (env, lp);
   cur_numcols = CPXgetnumcols (env, lp);
   for (i = 0; i < cur_numrows; i++) {
      printf (“Row %d:  Slack = %10f  Pi = %10f\n”, i, slack[i], pi[i]);
   }

   for (j = 0; j < cur_numcols; j++) {
      printf (“Column %d:  Value = %10f  Reduced cost = %10f\n”,
              j, x[j], dj[j]);
   }

   /* Finally, write a copy of the problem to a file. */

   status = CPXwriteprob (env, lp, “lpex6.sav”, NULL);
   if ( status ) {
      fprintf (stderr, “Failed to write LP to disk.\n”);
      goto TERMINATE;
   }

TERMINATE:

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, “CPXfreeprob failed, error code %d.\n”, status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, “Could not close CPLEX environment.\n”);
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, “%s”, errmsg);
      }
   }

   return (status);

}  /* END main */


/* This function builds by column the linear program:
      Maximize
       obj: x1 + 2 x2 + 3 x3
      Subject To
       c1: - x1 + x2 + x3 <= 20
       c2: x1 - 3 x2 + x3 <= 30
      Bounds
       0 <= x1 <= 40
      End
 */

#ifndef  CPX_PROTOTYPE_MIN
static int
populatebycolumn (CPXENVptr env, CPXLPptr lp)
#else
static int
populatebycolumn (env, lp)
CPXENVptr  env;
CPXLPptr   lp;
#endif
{
   int      status    = 0;
   double   obj[NUMCOLS];
   double   lb[NUMCOLS];
   double   ub[NUMCOLS];
   char     *colname[NUMCOLS];
   int      matbeg[NUMCOLS];
   int      matind[NUMNZ];
   double   matval[NUMNZ];
   double   rhs[NUMROWS];
   char     sense[NUMROWS];
   char     *rowname[NUMROWS];

   /* To build the problem by column, create the rows, and then
      add the columns. */

   CPXchgobjsen (env, lp, CPX_MAX);  /* Problem is maximization */

   /* Now create the new rows.  First, populate the arrays. */

   rowname[0] = “c1”;
   sense[0]   = ‘L’;
   rhs[0]     = 20.0;

   rowname[1] = “c2”;
   sense[1]   = ‘L’;
   rhs[1]     = 30.0;

   status = CPXnewrows (env, lp, NUMROWS, rhs, sense, NULL, rowname);
   if ( status )   goto TERMINATE;

   /* Now add the new columns.  First, populate the arrays. */

       obj[0] = 1.0;      obj[1] = 2.0;           obj[2] = 3.0;
```

```
    matbeg[0] = 0;     matbeg[1] = 2;         matbeg[2] = 4;

    matind[0] = 0;     matind[2] = 0;         matind[4] = 0;
    matval[0] = -1.0;  matval[2] = 1.0;       matval[4] = 1.0;

    matind[1] = 1;     matind[3] = 1;         matind[5] = 1;
    matval[1] = 1.0;   matval[3] = -3.0;      matval[5] = 1.0;

        lb[0] = 0.0;       lb[1] = 0.0;           lb[2]  = 0.0;
        ub[0] = 40.0;      ub[1] = CPX_INFBOUND;  ub[2]  = CPX_INFBOUND;

   colname[0] = “x1”; colname[1] = “x2”;      colname[2] = “x3”;

   status = CPXaddcols (env, lp, NUMCOLS, NUMNZ, obj, matbeg, matind,
                        matval, lb, ub, colname);
   if ( status )  goto TERMINATE;

TERMINATE:

   return (status);

}  /* END populatebycolumn */
```

Solving LP Problems with the Barrier Optimizer

This section tells you more about solving linear programming problems using the ILOG CPLEX *Barrier Optimizer.* (Chapter 7, *Solving Quadratic Programming Problems*, explains how to use the ILOG CPLEX Barrier Optimizer in convex quadratic problems.) It includes sections on:

- Identifying LPs for Barrier Optimization
- Interpreting the Barrier Log File
- Understanding Solution Quality from the Barrier LP Optimizer
- Overcoming Numerical Difficulties
- Diagnosing Barrier Optimizer Infeasibility

To use the ILOG CPLEX Barrier Optimizer in application development, you must hold a special, optional, development license. If you call barrier routines from the ILOG CPLEX Callable Library in your applications, your end users must be licensed for runtime or derived work. For more information about ILOG CPLEX licensing, contact your ILOG CPLEX representative.

Identifying LPs for Barrier Optimization

The ILOG CPLEX Barrier Optimizer is well suited to large, sparse problems. An alternative to the simplex optimizers, it exploits a primal-dual logarithmic barrier algorithm to generate a sequence of strictly positive primal and dual solutions to a problem. ILOG CPLEX finds the primal solutions, conventionally denoted (x, s), from the primal formulation:

Minimize $\boldsymbol{c}^T x$

subject to $\boldsymbol{A}x = \boldsymbol{b}$

with these bounds $x + s = u$ and $x \geq l$

where $\boldsymbol{A}$ is the constraint matrix, including slack and surplus variables; u is the upper and l the lower bounds on the variables.

Simultaneously, ILOG CPLEX automatically finds the dual solutions, conventionally denoted (y, z, w) from the corresponding dual formulation:

Maximize $b^T y - u^T w + l^T z$

subject to $\boldsymbol{A}^T y - w + z = \boldsymbol{c}$

with these bounds $w \geq 0$ and $z \geq 0$

All possible solutions maintain strictly positive primal solutions $(x - l, s)$ and strictly positive reduced costs (z, w) so that the value 0 (zero) forms a *barrier* for primal and dual variables within the algorithm.

ILOG CPLEX measures progress by the primal feasibility, dual feasibility, and duality gap at each iteration. To measure feasibility, ILOG CPLEX considers the accuracy with which the primal constraints ($\boldsymbol{A}x = b, x + s = u$) and dual constraints ($\boldsymbol{A}^T y + z - w = \boldsymbol{c}$) are satisfied. The optimizer stops when it finds feasible primal and dual solutions that are *complementary.* A complementary solution is one where the sums of the products $(x_j - l_j)z_j$ and $(u_j - x_j)z_j$ are within some tolerance of 0 (zero). Since each $(x_j - l_j)$, $(u_j - x_j)$, and z_j is strictly positive, the sum can be near zero only if each of the individual products is near zero. The sum of these products is known as the *complementarity* of the problem.

On each iteration of the barrier optimizer, ILOG CPLEX computes a matrix based on $\boldsymbol{AA^T}$ and then computes a *Cholesky factor* of it. This factored matrix has the same number of nonzeros on each iteration. The number of nonzeros in this matrix is influenced by the barrier *ordering* parameter. The particular ordering that is most effective depends on the platform (computer hardware and operating system) and the specific problem.

The ILOG CPLEX Barrier Optimizer is appropriate and often advantageous for large problems, for example, those with more than 1000 rows or columns. It is effective on problems with staircase structures or banded structures in the constraint matrix. It is also effective on problems with a small number of nonzeros per column.

Its performance is most dependent on these characteristics:

- the number of nonzeros in the Cholesky factor;
- the presence of dense columns, that is, columns with a relatively high number of nonzero entries.

To decide whether to use the barrier optimizer on a given problem, you should look at both these characteristics. (We explain how to check those characteristics later in this chapter in Step 2 and Step 3 on page 132.)

Barrier Simplex Crossover

Since many users prefer basic solutions because they can be used to restart optimization, the ILOG CPLEX Barrier Optimizer includes *basis crossover algorithms*. By default, the Interactive Barrier Optimizer `baropt` automatically invokes a primal crossover when the barrier algorithm terminates (unless termination occurs abnormally because of insufficient memory or numerical difficulties). Optionally, you can also execute barrier optimization with a dual crossover or with no crossover at all. The section Using the Barrier Optimizer in the Interactive Optimizer on page 132 explains how to control crossover in the Interactive Optimizer. From the Callable Library, use the routine `CPXhybbaropt()` with an argument to indicate crossover.

Differences between Barrier and Simplex Optimizers

The barrier optimizer and the simplex optimizers (primal and dual) are fundamentally different approaches to solving linear programming problems. The key differences between them have these implications:

- Simplex and barrier optimizers differ with respect to the *nature of solutions*.

  Simplex solutions are basic solutions. Barrier solutions are not. Consequently, when you use the barrier optimizer alone, you get no basis for advanced restarts. If you want to optimize the same or similar problems repeatedly, the barrier optimizer alone may not be appropriate.

  Also since a barrier solution is not a basic solution, no range information is available for sensitivity analysis when you use the barrier optimizer alone.

  Furthermore, barrier solutions tend to be midface solutions. In cases where multiple optima exist, barrier solutions tend to place the variables at values between their bounds, whereas in basic solutions from a simplex technique, the values of the variables are more likely to be at either their upper or their lower bound. While objective values will be the same, the nature of the solutions can be very different.

- Simplex and barrier optimizers have different *numerical properties*, sensitivity, and behavior. For example, the barrier optimizer is sensitive to the presence of unbounded optimal faces, whereas the simplex optimizers are not. As a result, problems that are numerically difficult for one method may be easier to solve by the other.

- Simplex and barrier optimizers have different *memory requirements*. Depending on the size of the Cholesky factor, the barrier optimizer can require significantly more memory than the simplex optimizers.
- Simplex and barrier optimizers work well on different *types of problems*. The barrier optimizer works well on problems where the AA^T remains sparse. Also, highly degenerate problems that pose difficulties for the primal or dual simplex optimizers may be solved quickly by the barrier optimizer. In contrast, the simplex optimizers will probably perform better on problems where the AA^T and the resulting Cholesky factor are relatively dense, though it is sometimes difficult to predict from the dimensions of the model when this will be the case.

Using the Barrier Optimizer

We have described how the ILOG CPLEX Barrier Optimizer finds primal and dual solutions from the primal and dual formulations of a model (see the section Identifying LPs for Barrier Optimization on page 130) , but you do not have to reformulate the problem yourself. The ILOG CPLEX Barrier Optimizer automatically creates the primal and dual formulations of the problem for you after you enter or read in the problem.

Using the Barrier Optimizer in the Interactive Optimizer

In the Interactive Optimizer, we recommend this way of working with the ILOG CPLEX Barrier Optimizer:

1. Enter the problem. You enter an LP problem to solve with the barrier optimizer just as you enter other LP problems. In the Interactive Optimizer, use the `enter` command to type in the problem data interactively, or use the `read` command to read a problem from a file in MPS, LP, or SAV format.
2. Check the number of nonzeros in the Cholesky factor. To do so, use the command `set barrier limits iterations 0`; then use the command `baropt stop`. These two commands together will make ILOG CPLEX count the number of nonzeros in the Cholesky factor but stop before it begins barrier iterations.
3. Check for dense columns. If you use the command `display problem histogram c`, ILOG CPLEX will show you the number of columns with nonzeros in the *un*preprocessed problem. If you want to see the density of columns in the processed, presolved, aggregated problem, do this:

   a. Write the preprocessed problem to a file with the file extension `.pre`; use the `write` command to do so.

   b. Read the file in with the `read` command.

   c. Display the histogram with its column option: `display problem histogram c.` You will be able to identify dense columns at a glance.

4. Use the `baropt` command to start optimization.

   The option `stop` tells ILOG CPLEX to stop with a nonbasis barrier solution. (Afterwards, you can apply the command `primopt` or `tranopt` explicitly to this nonbasis, barrier solution to cross over to a primal or dual basic solution.)

   The option `primopt` (or the shortcut `p`) tells ILOG CPLEX to cross over automatically to a basic solution using the primal simplex optimizer.

   The option `dualopt` (or the shortcut `d`) tells ILOG CPLEX to cross over to a basic solution using the dual simplex optimizer.

   If you specify no option, ILOG CPLEX assumes `dualopt`.

5. Check ILOG CPLEX progress. Use the command `set barrier display` to change the level of information displayed on the screen or logged to a file.

Using the Barrier Optimizer in the Component Libraries

Initialize the ILOG CPLEX environment, create the problem object, and populate the problem object, as explained in *Creating an Application with CPLEX Concert Technology Library* on page 29 and *Using the Callable Library in an Application* on page 57.

◆ In the Concert Technology Library, use the method:

  - `cplex.setRootAlgorithm(IloCplex::Barrier)` to invoke the CPLEX Barrier Optimizer without crossover.
  - `cplex.setRootAlgorithm(IloCplex::BarrierPrimal)` to invoke the CPLEX Barrier Optimizer with primal crossover.
  - `cplex.setRootAlgorithm(IloCplex::BarrierDual)` to invoke the CPLEX Barrier Optimizer with dual crossover.

◆ In the Callable Library, use the routine:

  - `CPXhybbaropt()` to invoke the ILOG CPLEX Barrier Optimizer with crossover.
  - `CPXbaropt()` to invoke the ILOG CPLEX Barrier Optimizer without crossover.

Special Options

In addition to the parameters available for other ILOG CPLEX LP optimizers, there are also parameters to control the ILOG CPLEX Barrier Optimizer. In the Interactive Optimizer, to see a list of the parameters specific to the ILOG CPLEX Barrier Optimizer, use the command `set barrier`.

Controlling Crossover

In the Concert Technology Library, crossover is specified in the method that invokes the optimizer.

In the Interactive Optimizer, options to the `baropt` command control whether the ILOG CPLEX Barrier Optimizer stops with a nonbasic solution or crosses over to a simplex optimizer to generate a basic solution. Table 4.8 summarizes those options to the `baropt` command in the Interactive Optimizer.

Table 4.8 *Options to the Barrier Optimizer to Control Crossover*

Option	Purpose
(no option)	ILOG CPLEX assumes `primopt` option
`stop`	ILOG CPLEX stops optimization with a nonbasic, barrier solution
`primopt`	After barrier optimization, ILOG CPLEX uses primal crossover
`dualopt`	After barrier optimization, ILOG CPLEX uses dual crossover

Table 4.9 shows you the corresponding routines from the Callable Library.

Table 4.9 *Routines of the Callable Library to Control Crossover*

Option	Callable Library routine
(no option)	`CPXhybbaropt (env, lp, CPX_ALG_PRIMAL)`
`stop`	`CPXbaropt (env, lp)`
`primopt`	`CPXhybbaropt (env, lp, CPX_ALG_PRIMAL)`
`dualopt`	`CPXhybbaropt (env, lp, CPX_ALG_DUAL)`

Using VEC File Format

When you use the ILOG CPLEX Barrier Optimizer with no crossover (for example, with the command `baropt stop`), you can save the primal and dual variable values and their associated reduced cost and dual values in a VEC-format file. You can then read that VEC file into ILOG CPLEX before you initiate a crossover at a later time. After you read a VEC file into ILOG CPLEX, all three optimizers—primal simplex, dual simplex, and barrier simplex—automatically invoke crossover.

Even if you have set the advanced basis indicator to `no` (meaning that you do not intend to start from an advanced basis), ILOG CPLEX automatically resets the indicator to `yes` when it reads a VEC file. If you turn off the advanced basis indicator after reading a VEC file, then the simplex optimizers will honor this setting and will not initiate crossover.

Interpreting the Barrier Log File

Like the ILOG CPLEX Simplex Optimizers, the ILOG CPLEX Barrier Optimizer records information about its progress in a log file as it works. Some users find it helpful to keep a new log file for each session. By default, ILOG CPLEX records information in a file named `cplex.log`. In the:

- Interactive Optimizer, use the command `set logfile filename` to change the name of the log file.
- Callable Library, use the routine `CPXsetlogfile()` with arguments to indicate the log file.

You can control the level of information about barrier optimization that ILOG CPLEX records in the log file.

- **Level one**, the default, includes the usual and customary information, explained in greater detail later in this section.
- **Level two**, rarely needed, gives information about the automatically computed barrier column-nonzeros parameter and provides diagnostic detail for ILOG CPLEX technical support. To set level two, in the:
  - Interactive Optimizer, use the command `set barrier display 2`.
  - From the Callable Library, set the parameter `CPX_PARAM_BARDISPLAY`.
- To turn off progress information entirely, use the value `0` (zero) in the command or routine.

To give you an idea about a barrier log file, here is the log file for a pure barrier optimization (that is, the `baropt` command with the `stop` option) at display level one (the default).

```
Tried aggregator 1 time.
LP Presolve eliminated 9 rows and 11 columns.
Aggregator did 6 substitutions.
Reduced LP has 12 rows, 15 columns, and 38 nonzeros.
Presolve time =    0.00 sec.
Number of nonzeros in lower triangle of A*A' = 26
Using Approximate Minimum Degree ordering
Total time for automatic ordering = 0.00 sec.
Summary statistics for Cholesky factor:
  Rows in Factor            = 12
  Integer space required    = 12
  Total non-zeros in factor = 78
  Total FP ops to factor    = 650
 Itn      Primal Obj        Dual Obj  Prim Inf Upper Inf  Dual Inf
   0  -1.3177911e+01  -1.2600000e+03  6.55e+02  0.00e+00  3.92e+01
   1  -4.8683118e+01  -5.4058675e+02  3.91e+01  0.00e+00  1.18e+01
   2  -1.6008142e+02  -3.5969226e+02  1.35e-13  7.11e-15  5.81e+00
   3  -3.5186681e+02  -6.1738305e+02  1.59e-10  1.78e-15  5.16e-01
   4  -4.5808732e+02  -4.7450513e+02  5.08e-12  1.95e-14  4.62e-02
   5  -4.6435693e+02  -4.6531819e+02  1.66e-12  1.27e-14  1.59e-03
   6  -4.6473085e+02  -4.6476678e+02  5.53e-11  2.17e-14  2.43e-15
   7  -4.6475237e+02  -4.6475361e+02  5.59e-13  2.99e-14  2.19e-15
   8  -4.6475312e+02  -4.6475316e+02  1.73e-13  1.55e-14  1.17e-15
   9  -4.6475314e+02  -4.6475314e+02  1.45e-13  2.81e-14  2.17e-15

Barrier - Optimal:  Objective =   -4.6475314194e+02
Solution time =    0.01 sec.  Iterations = 9
```

Preprocessing in the Log File

The opening lines of that log file record information about *preprocessing* by the ILOG CPLEX presolver and aggregator. After those preprocessing statistics, the next line records the number of *nonzeros in the lower triangle* of a particular matrix, AA^T, denoted `A*A'` in the log file.

Nonzeros in Lower Triangle of AA^T in the Log File

The number of nonzeros in the lower triangle of AA^T gives an early indication of how long each barrier iteration will take. The larger this number, the more time each barrier iteration requires. If this number is close to 50% of the square of the number of rows, then the problem may contain dense columns that are not being detected. In that case, examine the histogram of column counts; then consider setting the barrier column-nonzeros parameter to a value that enables ILOG CPLEX to treat more columns as being dense.

Ordering-Algorithm Time in the Log File

After the number of nonzeros in the lower triangle of AA^T, ILOG CPLEX records the time required by the ordering algorithm. (The ILOG CPLEX Barrier Optimizer offers you a choice of four ordering algorithms, explained in *Choosing an Ordering Algorithm* on page 142.) This line in the log file verifies that ILOG CPLEX is using the order you chose.

Cholesky Factor in the Log File

After the time required by the ordering algorithm, ILOG CPLEX records information about the *Cholesky factor.* ILOG CPLEX computes this matrix on each iteration. The *number of rows* in the Cholesky factor represents the number after preprocessing. The *size of the dense window* indicates how dense the factored matrix is. If the size of the dense window is large with respect to the number of rows, then the Cholesky factor is dense, and consequently, the ILOG CPLEX Barrier Optimizer will require more time per iteration.

The next line of information about the Cholesky factor—integer space required—indicates the amount of memory needed to store the *sparsity pattern* of the factored matrix. If this number is low, then the factor can be computed more quickly than when the number is high.

Information about the Cholesky factor ends with the number of nonzeros in the factored matrix. This number is directly related to the time required per iteration of the ILOG CPLEX Barrier Optimizer. In fact, the difference between this number and the number of nonzeros in $\boldsymbol{A}\boldsymbol{A}^{\boldsymbol{T}}$ indicates the *fill-level* of the Cholesky factor. If the fill-level is large, consider an alternate ordering algorithm.

Iteration Progress in the Log File

After the information about the Cholesky factor, the log file records progress at each iteration. It records both *primal* and *dual objectives* (as `Primal Obj` and `Dual Obj`) per iteration.

It also records absolute infeasibilities per iteration. Internally, the ILOG CPLEX Barrier Optimizer treats inequality constraints as equality constraints with added slack and surplus variables. Consequently, primal constraints in a problem are written as $\boldsymbol{A}x = b$ and $x + s = u$, and the dual constraints are written as $\boldsymbol{A}^{\boldsymbol{T}}y + z - w = c$. As a result, in the log file, the infeasibilities represent *norms*, as summarized in Table 4.10.

Table 4.10 *Infeasibilities and Norms in the Log File of a Barrier Optimization*

Infeasibility	**In log file**	**Norm**
primal	`Prim Inf`	$\|b - \mathbf{A}x\|$
upper	`Upper Inf`	$\|u - (x + s)\|$
dual	`Dual Inf`	$\|c - y\mathbf{A} - z + w\|$

If solution values are large in absolute value, then the infeasibilities may appear inordinately large because they are recorded in the log file in absolute terms. The optimizer uses *relative* infeasibilities as termination criteria.

Infeasibility Ratio in the Log File

If you are using one of the barrier infeasibility algorithms available in the ILOG CPLEX Barrier Optimizer (that is, in the Interactive Optimizer you have used the command `set barrier algorithm 1` or `set barrier algorithm 2` or from the Callable

Library, you used the routine `CPXsetintparam()` to set the parameter `CPX_PARAM_BARALG` to the value `1` or `2`), then ILOG CPLEX records an additional column of output titled `Inf Ratio`, the infeasibility ratio. This ratio, always positive, is a measure of progress for that particular algorithm. In a problem with an optimal solution, you will see this ratio increase to a large number. In contrast, in a problem that is primal infeasible or dual infeasible, this ratio will decrease to a very small number.

Understanding Solution Quality from the Barrier LP Optimizer

When ILOG CPLEX successfully solves a problem with the ILOG CPLEX Barrier Optimizer, it reports the optimal objective value and solution time in a log file, as it does for other LP optimizers.

Because barrier solutions (prior to crossover) are not basic solutions, certain solution statistics associated with basic solutions are not available for a strictly barrier solution. For example, reduced costs and dual values are available for strictly barrier LP solutions, but range information about them is not.

To help you evaluate the quality of a barrier solution more readily, ILOG CPLEX offers a special display of information about barrier solution quality. To display this information in the Interactive Optimizer, use the command `display solution quality` after optimization. When using the Component Libraries, use the method `cplex.getQuality()` or use the routines `CPXgetintquality()` for integer information and `CPXgetdblquality()` for double-valued information.

Table 4.11 *Barrier Solution Quality Display*

Item	Meaning
primal objective	primal objective value $c^T x$
dual objective	dual objective value $b^T y - u^T w + l^T z$
duality gap	difference between primal and dual objectives
complementarity	sum of column and row complementarity
column complementarity (total)	sum of $\lvert(x_j - l_j)\bullet z_j\rvert + \lvert(u_j - x_j)\bullet w_j\rvert$
column complementarity (max)	maximum of $\lvert(x_j - l_j)\bullet z_j\rvert$ and $\lvert(u_j - x_j)\bullet w_j\rvert$ over all variables
row complementarity (total)	sum of $\lvert slack_i \bullet y_i\rvert$
row complementarity (max)	maximum of $\lvert slack_i \bullet y_i\rvert$
primal norm |x| (total)	sum of absolute values of all primal variables

Table 4.11 *Barrier Solution Quality Display (Continued)*

Item	**Meaning**
primal norm |x| (max)	maximum of absolute values of all primal variables
dual norm |rc| (total)	sum of absolute values of all reduced costs
dual norm |rc| (max)	maximum of absolute values of all reduced costs
primal error (Ax = b) (total, max)	total and maximum error in satisfying primal equality constraints
dual error (A'pi + rc = c) (total, max)	total and maximum error in satisfying dual equality constraints
primal x bound error (total, max)	total and maximum error in satisfying primal lower and upper bound constraints
primal slack bound error (total, max)	total and maximum violation in slack variables
dual pi bound error (total, max)	total and maximum violation with respect to zero of dual variables on inequality rows
dual rc bound error (total, max)	total and maximum violation with respect to zero of reduced costs
primal normalized error (Ax = b) (max)	accuracy of primal constraints
dual normalized error (A'pi + rc = c) (max)	accuracy of dual constraints

Table 4.11 lists the items ILOG CPLEX displays and explains their meaning. In the solution quality display, the term pi refers to dual solution values, that is, the *y* values in the conventional barrier problem-formulation. The term rc refers to reduced cost, that is, the difference $z - w$ in the conventional barrier problem-formulation. Other terms are best understood in the context of primal and dual LP formulations.

Normalized errors, for example, represent the accuracy of satisfying the constraints while considering the quantities used to compute $\boldsymbol{A}x$ on each row and $y^T\boldsymbol{A}$ on each column. In the primal case, for each row, we consider the nonzero coefficients and the x_j values used to compute $\boldsymbol{A}x$. If these numbers are large in absolute value, then it is acceptable to have a larger absolute error in the primal constraint.

Similar reasoning applies to the dual constraint.

If ILOG CPLEX returned an optimal solution, but the primal error seems high to you, the primal *normalized* error should be low, since it takes into account the scaling of the problem and solution.

After a simplex optimization—whether primal, dual, or network—or after a crossover, the display command will display information related to the quality of the *simplex* solution.

Tuning Barrier Optimizer Performance

Naturally, the default parameter settings for the ILOG CPLEX Barrier Optimizer work best on most problems. However, you can tune several algorithmic parameters to improve performance or to overcome numerical difficulties. These parameters are described in the sections:

- Out-of-Core Barrier: Letting the Optimizer Use Disk for Storage
- Preprocessing: the Presolver and Aggregator
- Detecting and Eliminating Dense Columns
- Choosing an Ordering Algorithm
- Using a Starting-Point Heuristic

In addition, several parameters set termination criteria. With them, you control when ILOG CPLEX stops optimization.

You can also control convergence tolerance—another factor that influences performance. Convergence tolerance determines how nearly optimal a solution ILOG CPLEX must find: tight convergence tolerance means ILOG CPLEX must keep working until it finds a solution very close to the optimal one; loose tolerance means ILOG CPLEX can return a solution within a greater range of the optimal one and thus stop calculating sooner.

Performance of the ILOG CPLEX Barrier Optimizer is most highly dependent on the *size of the Cholesky factor* computed at each iteration. When you adjust barrier parameters, always check their impact on the size of the Cholesky factor. At default output settings, this size is reported at the beginning of each barrier optimization in the log file, as we explain in *Cholesky Factor in the Log File* on page 137.

Another important performance issue is the presence of *dense columns*. By a dense column, we mean that a given variable appears in a relatively large number of rows. You can check column density as we suggest in Step 3 on page 132. We also say more about column density in *Detecting and Eliminating Dense Columns* on page 142.

In adjusting parameters, you may need to experiment to find beneficial settings because the precise effect of parametric changes will depend on the nature of your LP problem as well as your platform (hardware, operating system, compiler, etc.). Once you have found satisfactory parametric settings, keep them in a parameter specification file for re-use, as explained in *Saving a Parameter Specification File* on page 339.

Out-of-Core Barrier: Letting the Optimizer Use Disk for Storage

Under default settings, the CPLEX Barrier Optimizer will do all of its work using central memory (variously referred to also as RAM, core, or physical memory). For models too large to solve in the central memory on your computer, or in cases where it is simply not desired to use this much memory, it is possible to instruct the barrier optimizer to use disk for part of the working storage it needs, specifically the Cholesky factorization. Since disk is slower than central memory, there may be some lost performance by making this choice on models that could be solved entirely in central memory, but the out-of-core feature in the CPLEX Barrier Optimizer is designed to make this as efficient as possible. It generally will be far more effective than relying on the operating system's virtual memory (swap space).

To activate out-of-core Barrier:

◆ In the Interactive Optimizer, use the command: `set barrier outofcore yes`.

◆ When using the Component Libraries, set the parameter `IloCplex::BarOOC` or `CPX_PARAM_BAROOC` to `1`.

Even when out-of-core Barrier is activated, the factorization will stay in central memory unless its size exceeds the value of the Working Memory parameter. The default for this parameter is 128, meaning 128 megabytes.

To select a different threshold for use of disk working storage, say 32 megabytes:

◆ In the Interactive Optimizer, use the command: `set workmem  32`.

◆ When using the Component Libraries, set the parameter `IloCplex::WorkMem` or `CPX_PARAM_WORKMEM`.

When Barrier is being run out-of-core, the location of disk storage is controlled by the Working Directory parameter. For example, if you wish to use the directory `/tmp/mywork` for this purpose, where this directory already exists at the start of the CPLEX Barrier run:

◆ In the Interactive Optimizer, use the command `set workdir /tmp/mywork`.

◆ When using the Component Libraries, set the parameter `IloCplex::WorkDir` or `CPX_PARAM_WORKDIR` to be the string '`/tmp/mywork`'.

Preprocessing: the Presolver and Aggregator

For best performance of the ILOG CPLEX Barrier Optimizer, preprocessing should almost always be on. That is, we recommend that you use the default setting where the presolver and aggregator are active. While they may use more memory, they also reduce the problem, and problem reduction is crucial to barrier optimizer performance. In fact, reduction is so important that even when you turn off preprocessing, ILOG CPLEX still applies minimal presolving before barrier optimization.

For problems that contain linearly dependent rows, it is a good idea to turn on the *preprocessing dependency parameter.* (By default, it is off.) This dependency checker may add some preprocessing time, but it can detect and remove linearly dependent rows to improve overall performance.

To turn on the preprocessing dependency parameter:

- In the Interactive Optimizer, use the command `set preprocessing dependency 1`.
- When using the Component Libraries, set the parameter `IloCplex::DepInd` or `CPX_PARAM_DEPIND`.

Detecting and Eliminating Dense Columns

Dense columns can significantly degrade barrier optimizer performance. (A dense column is one in which a given variable appears in many rows.) For that reason, we recommend that after you enter or read a problem for barrier optimization, you check it for dense columns by inspecting its column histogram after preprocessing, as in Step 3 on page 132.

In fact, when a few dense columns are present in a problem, it is often effective to reformulate the problem to remove those dense columns from the model.

Otherwise, you can control whether ILOG CPLEX perceives columns as dense by setting the column nonzeros parameter. At its default setting, ILOG CPLEX calculates an appropriate value for this parameter automatically. However, if your problem contains one (or a few) dense columns that remain undetected at the default setting, you can adjust this parameter yourself to help ILOG CPLEX detect it (or them). For example, in a large problem in which one column contains forty entries while the other columns contain less than five entries, you will benefit by setting the column nonzeros parameter to 30. This setting allows ILOG CPLEX to recognize that column as dense and thus invoke techniques to handle it.

To set the column nonzeros parameter:

- In the Interactive Optimizer, use the command `set barrier colnonzeros i`, substituting a positive integer for `i`.
- When using the Component Libraries, set the parameter `IloCplex::BarColNz` or `CPX_PARAM_BARCOLNZ`.

Once ILOG CPLEX detects a dense column, it takes steps to eliminate it.

Choosing an Ordering Algorithm

ILOG CPLEX offers several different algorithms in the CPLEX Barrier Optimizer for ordering the rows of a matrix:

- automatic, the default, indicated by the value `0`;
- approximate minimum degree (AMD), indicated by the value `1`;
- approximate minimum fill (AMF) indicated by the value `2`;
- nested dissection (ND) indicated by the value `3`.

The log file, as we explain in *Ordering-Algorithm Time in the Log File* on page 136, records the time spent by the ordering algorithm in a barrier optimization, so you can experiment with different ordering algorithms and compare their performance on your problem.

Automatic ordering, the default option, will usually be the best choice. This option attempts to choose the most effective of the available ordering methods, and it usually results in the best order. It may require more time than the other settings. The ordering time is usually small relative to the total solution time, and a better order can lead to a smaller total solution time. In other words, a change in this parameter is unlikely to improve performance very much.

The AMD algorithm provides good quality order within moderate ordering time. AMF usually provides better order than AMD (usually 5-10% smaller factors) but it requires somewhat more time (10-20% more). ND often produces significantly better order than AMD or AMF. Ten-fold reductions in runtimes of the ILOG CPLEX Barrier Optimizer have been observed with it on some problems. However, ND sometimes produces worse order, and it requires much more time.

To change from one ordering algorithm to another:

◆ In the Interactive Optimizer, use the command `set barrier ordering i`, substituting a value (`0`, `1`, `2`, or `3`) for `i`.

◆ When using the Component Libraries, set the parameter `IloCplex::BarOrder` or `CPX_PARAM_BARORDER`.

Using a Starting-Point Heuristic

ILOG CPLEX supports several different heuristics to compute the starting point for the ILOG CPLEX Barrier Optimizer. Table 4.12 summarizes the parameter values to indicate which starting-point heuristic to use.

Table 4.12 *Parameter Values for Starting-Point Heuristics*

Value	Heuristic
1	dual is 0 (default)
2	estimate dual
3	average primal estimate, dual 0
4	average primal estimate, estimate dual

For most problems the default works well. Indeed, changing the starting-point heuristic may even worsen performance overall. However, if you are using the dual preprocessing option (for example, `set preprocessing dual 1`), then one of the other heuristics for computing a starting point may perform better than the default.

To change the starting point heuristic:

- ◆ In the Interactive Optimizer, use the command `set barrier startalg i`, substituting a value for `i`.
- ◆ When using the Component Libraries, set the parameter `IloCplex::BarStartAlg` or `CPX_PARAM_BARSTARTALG`.

Overcoming Numerical Difficulties

As we noted in *Differences between Barrier and Simplex Optimizers* on page 131, the algorithms in the barrier optimizer have very different numerical properties from those in the simplex optimizer. While the barrier optimizer is often extremely fast, particularly on very large problems, numerical difficulties occasionally arise with it in certain classes of problems. For that reason, we recommend that you run simplex optimizers in conjunction with the barrier optimizer to verify solutions. At its default settings, the ILOG CPLEX Barrier Optimizer always crosses over after a barrier solution to a simplex optimizer, so this verification occurs automatically.

Difficulties in the Quality of Solution

Understanding Solution Quality from the Barrier LP Optimizer on page 138 lists the items that ILOG CPLEX displays about the quality of a barrier solution. If the ILOG CPLEX Barrier Optimizer terminates its work with a solution that does not meet your quality requirements, you can adjust parameters that influence the quality of a solution. Those adjustments affect the choice of ordering algorithm, the choice of barrier algorithm, the limit on barrier corrections, and the choice of starting-point heuristic—topics introduced in *Tuning Barrier Optimizer Performance* on page 140 and recapitulated here in the following subsections.

Change the Ordering Algorithm

As we explain about tuning performance in *Choosing an Ordering Algorithm* on page 142, you can choose one of several ordering algorithms to use in the ILOG CPLEX Barrier Optimizer. To improve the quality of a solution in some problems, change the ordering algorithm, as suggested on page 143.

Change the Barrier Algorithm

The ILOG CPLEX Barrier Optimizer implements the algorithms listed in Table 4.13. The default option invokes option `3` for LPs and option `1` for MIPs where the ILOG CPLEX Barrier Optimizer is used on the subproblems. Naturally, the default is the fastest for most problems, but it may not work well on problems that are primal infeasible or dual infeasible. Options `1` and `2` in the ILOG CPLEX Barrier Optimizer implement a barrier algorithm that also detects infeasibility. (They differ from each other in how they compute a starting point.) Though they are slower than the default option, in a problem demonstrating numerical difficulties, they may eliminate the numerical difficulties and thus improve the quality of the solution.

Table 4.13 *Values of the Parameter to Choose the Algorithm in the Barrier Optimizer*

Value	Meaning
`0`	default
`1`	algorithm starts with infeasibility estimate
`2`	algorithm starts with infeasibility constant
`3`	standard barrier algorithm

To change the barrier algorithm:

- ◆ In the Interactive Optimizer, use the command `set barrier algorithm i`.
- ◆ When using the Component Libraries, set the parameter `IloCplex::BarAlg` or `CPX_PARAM_BARALG`.

Change the Limit on Barrier Corrections

The default barrier algorithm in the ILOG CPLEX Barrier Optimizer computes an estimate of the maximum number of *centering corrections* that ILOG CPLEX should make on each iteration. You can see this computed value by setting barrier display level two, as explained in *Interpreting the Barrier Log File* on page 135, and checking the value of the parameter to limit corrections. (Its default value is `-1`.) If you see that the current value is `0` (zero), then you should experiment with greater settings. Setting this parameter to a value greater than `0` may improve numerical performance, but there may also be an increase in computation time.

To set the parameter to limit barrier corrections:

- ◆ In the Interactive Optimizer use the command `set barrier limits corrections i`, substituting an integer greater than zero but less than or equal to ten for `i`.
- ◆ When using the Component Libraries, set the parameter `IloCplex::BarMaxCor` or `CPX_PARAM_BARMAXCOR`.

Choose a Different Starting-Point Heuristic

As we explained in *Using a Starting-Point Heuristic* on page 143, the default starting-point heuristic works well for most problems suitable to barrier optimization, and in fact, changing the starting-point heuristic can worsen performance. However, if you are preprocessing your problem as dual (for example, in the Interactive Optimizer you issued the command `set preprocessing dual`), then a different starting-point heuristic may perform better than the default. To change the starting-point heuristic, see Table 4.12 on page 143.

Difficulties during Optimization

Numerical difficulties can degrade performance of the ILOG CPLEX Barrier Optimizer or even prevent convergence toward a solution. There are several possible sources of numerical difficulties:

- elimination of too many dense columns may cause numerical instability;
- tight convergence tolerance may aggravate small numerical inconsistencies in a problem;
- unbounded optimal faces may remain undetected and thus prevent convergence.

The following subsections offer guidance about overcoming those difficulties.

Numerical Instability Due to Elimination of Too Many Dense Columns

Detecting and Eliminating Dense Columns on page 142 explains how to change parameters to encourage ILOG CPLEX to detect and eliminate as many dense columns as possible. However, in some problems, if ILOG CPLEX removes too many dense columns, it may cause numerical instability.

You can check how many dense columns ILOG CPLEX removes by looking at the preprocessing statistics at the beginning of the log file, as we explained in *Preprocessing in the Log File* on page 136. If you observe that the removal of too many dense columns results in numerical instability in your problem, then increase the column nonzeros parameter.

The default value of the column nonzeros parameter is `0` (zero); that value tells ILOG CPLEX to calculate the parameter automatically.

To see the current value of the column nonzeros parameter—either one you have set or one ILOG CPLEX has automatically calculated—you need to look at the level two display. To see the level two display:

- In the Interactive Optimizer, use the command `set barrier display 2`.
- From the Callable Library, set the parameter `CPX_PARAM_BARDISPLAY`.

Either alternative will record level two information in the log file, where you can see the current value of the column nonzeros parameter.

If you determine that the current value of the column nonzeros parameter is inappropriate for your problem and thus tells ILOG CPLEX to remove too many dense columns, then you can *increase* the parameter to keep the number of dense columns removed *low*. In the Interactive Optimizer, use the command `set barrier colnonzeros i`, substituting a larger value for `i`. When using the Component Libraries, set the parameter `IloCplex::BarColNz` or `CPX_PARAM_BARCOLNZ`.

Small Numerical Inconsistencies and Tight Convergence Tolerance

If your problem contains small numerical inconsistencies, it may be difficult for the ILOG CPLEX Barrier Optimizer to achieve a satisfactory solution at the default setting of the complementarity convergence tolerance. In such a case, you should increase that tolerance to a value greater than its default, $1e^{-8}$.

To increase complementarity convergence tolerance:

- In the Interactive Optimizer, use the command `set barrier convergetol i`, substituting a greater than or equal to *1e*$^{-10}$ for `i`.
- When using the Component Libraries, set the parameter `IloCplex::BarEpComp` or `CPX_PARAM_BAREPCOMP`.

Unbounded Variables and Unbounded Optimal Faces

An *unbounded optimal face* occurs in an LP that contains a sequence of optimal solutions, all with the same value for the objective function and unbounded variable values. The ILOG CPLEX Barrier Optimizer will fail to terminate normally if an undetected unbounded optimal face exists.

Normally, the ILOG CPLEX Barrier Optimizer uses its barrier growth parameter to detect such conditions. If this parameter is increased beyond it default value, the ILOG CPLEX Barrier Optimizer will be less likely to determine that the problem has an unbounded optimal face and more likely to encounter numerical difficulties.

Consequently, you should change the barrier growth parameter *only* if you find that the ILOG CPLEX Barrier Optimizer is terminating its work before it finds the true optimum because it has falsely detected an unbounded face.

Furthermore, if you know that all the variables in your problem have a finite upper bound, then you should set an upper bound on all previously unbound variables in your problem.

To set an upper bound on unbound variables:

- In the Interactive Optimizer, use the command `set barrier limits varupper i`, substituting your known upper bound for `i`.
- When using the Component Libraries, set the parameter `IloCplex::BarVarUp` or `CPX_PARAM_BARVARUP`.

ILOG CPLEX will then use that upper bound to temporarily set a bound on any previously unbound variables.

Difficulties with Unbounded Problems

ILOG CPLEX detects unbounded problems in either of two ways:

- either it finds a solution with small complementarity that is not feasible for either the primal or the dual formulation of the problem;
- or the iterations tend toward infinity with the objective value becoming very large in absolute value.

The ILOG CPLEX Barrier Optimizer stops when the absolute value of either the primal or dual objective exceeds the object-range parameter.

If you increase the value of the object-range parameter, then the ILOG CPLEX Barrier Optimizer will iterate more times before it decides that the current problem suffers from an unbounded objective value.

If you know that your problem has *large objective values*, consider increasing the object-range parameter. In the Interactive Optimizer, use the command `set barrier limits objrange i`, substituting a large positive value for `i`. When using the Component Libraries, set the parameter `IloCplex::BarObjRng` or `CPX_PARAM_BAROBJRNG`.

Also if you know that your problem has *large objective values*, consider changing the algorithm that the ILOG CPLEX Barrier Optimizer is using from the default to one of the alternatives. To change the algorithm:

- In the Interactive Optimizer, use the command `set barrier algorithm i`, substituting a value from Table 4.13 on page 145.
- When using the Component Libraries, set the parameter `IloCplex::BarAlg` or `CPX_PARAM_BARALG`.

Diagnosing Barrier Optimizer Infeasibility

When the ILOG CPLEX Barrier Optimizer terminates and reports an infeasible solution, all the usual solution information is available. However, the solution values, reduced costs, and dual variables reported then do not correspond to a basis; hence, that information does not have the same meaning as the corresponding output from the ILOG CPLEX simplex optimizers.

Actually, since the ILOG CPLEX Barrier Optimizer works in a single phase, all reduced costs and dual variables are calculated in terms of the *original* objective function.

If the ILOG CPLEX Barrier Optimizer reports to you that a problem is infeasible, but you still need a basic solution for the problem, use the primal simplex optimizer. For example, in the Interactive Optimizer, use the command `primopt`. ILOG CPLEX will then use the solution provided by the barrier optimizer to determine a starting basis for the primal simplex optimizer. When the primal simplex optimizer finishes its work, you will have an infeasible basic solution for further infeasibility analysis.

If the default algorithm in the ILOG CPLEX Barrier Optimizer determines that your problem is primal infeasible or dual infeasible, then try the alternate algorithms in the barrier optimizer. These algorithms, though slower than the default, are better at detecting primal and dual infeasibility. To change the algorithm:

- In the Interactive Optimizer, use the command `set barrier algorithm i`, substituting a value from Table 4.13 on page 145.
- When using the Component Libraries, set the parameter `IloCplex::BarAlg` or `CPX_PARAM_BARALG`.

In *Finding a Set of Irreducibly Inconsistent Constraints* on page 116, we explained how to invoke the infeasibility finder on a solution basis found by one of the simplex optimizers. If you are using the *pure* barrier optimizer (that is, with no crossover to a simplex optimizer), then it will not generate a basis on which you can call the infeasibility finder to analyze your constraints and locate an IIS. Consequently, if you are interested in finding an IIS for your problem, you should invoke the ILOG CPLEX Barrier Optimizer with crossover, as explained in *Controlling Crossover* on page 134.

C H A P T E R

5

Solving Mixed Integer Programming Problems

The ILOG CPLEX *Mixed Integer Optimizer* enables you to solve models in which one or more variables must be restricted to integer solution values. This chapter tells you more about optimizing mixed integer programming (MIP) problems with ILOG CPLEX. It includes sections on:

- Sample: Stating a MIP Problem
- Considering Preliminary Issues
- Using the Mixed Integer Optimizer
- Using Sensitivity Information in a MIP
- Using Special Ordered Sets (SOS)
- Using Semi-Continuous Variables
- Progress Reports: Interpreting the Node Log
- Troubleshooting MIP Performance Problems
- Example: Optimizing a Basic MIP Problem
- Example: Reading a MIP Problem from a File
- Example: Using SOS and Priority

To use the ILOG CPLEX Mixed Integer Optimizer in application development, your development license must include the MIP option. If you call MIP routines from the Concert Technology or Callable Libraries in your applications, your end-users' runtime (derivative work) licenses must also include the MIP option. For more information about ILOG CPLEX licensing, contact your ILOG CPLEX representative.

Sample: Stating a MIP Problem

A mixed integer programming (MIP) problem may consist of both integer and continuous variables. The integer variables may be restricted to the values *0* (zero) and *1* (one), in which case they are referred to as *binary* variables. Or they may take on any integer values, in which case they are referred to as *general* integer variables. A variable that may take either the value 0 or a value between a lower and an upper bound is referred to as *semi-continuous*. A semi-continuous variable that is restricted to integer values is referred to as *semi-integer.* (Continuous variables in a mixed integer programming problem are *not* restricted to integer values.) The following illustrates a mixed integer programming problem, which is solved in the example program `ilomipex1.cpp` / `mipex1.c`, discussed later in this chapter:

$$
\begin{array}{lrcccccccccc}
\text{Maximize} & & x_1 & + & 2x_2 & + & 3x_3 & + & x_4 & & \\
\text{subject to} & - & x_1 & + & x_2 & + & x_3 & + & 10x_4 & \leq & 20 \\
 & & x_1 & - & 3x_2 & + & x_3 & & & \leq & 30 \\
 & & & & x_2 & & & - & 3.5x_4 & = & 0 \\
\text{with these bounds} & & 0 & \leq & x_1 & \leq & 40 & & & & \\
 & & 0 & \leq & x_2 & \leq & +\infty & & & & \\
 & & 0 & \leq & x_3 & \leq & +\infty & & & & \\
 & & 2 & \leq & x_4 & \leq & 3 & & & & \\
 & & & & x_4 & \text{integer} & & & & &
\end{array}
$$

Considering Preliminary Issues

When you are optimizing a MIP, there are a few preliminary issues that you need to consider to get the most out of ILOG CPLEX. The following sections cover such topics as entering variable type, displaying MIPs in the Interactive Optimizer, determining the problem type, and switching to relaxed versions of your problem.

Entering MIP Problems

You enter MIPs into ILOG CPLEX in the same way as LPs, as explained in *Put Data in the Problem Object* on page 58, with this additional consideration: you need to indicate which variables are binary, general integer, semi-continuous, and semi-integer, and which are contained in special ordered sets (SOS).

Concert Technology Library users can specify this information by passing a type value to the appropriate constructor when creating the variable. Use `IloNumVar::Bool` for binary variables, `IloNumVar::Int` for general integer variables, `IloSemiContVar` for semi-continuous variables, and `IloSemiContVar::Int` for semi-integer variables. SOS variables are created as instances of the class `IloSOS1` or `IloSOS2`.

Callable Library users can specify this information through the `CPXcopyctype()` routine.

In the Interactive Optimizer, to indicate binary integers in the context of the `enter` command, type `binaries` on a separate line, followed by the designated binary variables. To indicate general integers, type `generals` on a separate line, followed by the designated general variables. To indicate semi-continuous variables, type `semi-continuous` on a separate line, followed by the designated variables. Semi-integer variables are indicated by being specified as **both** general integer and semi-continuous. The order of these three sections does not matter. To enter the general integer variable of the *Sample: Stating a MIP Problem* on page 152, you type this:

```
generals
x4
```

You may also read MIP data in from a formatted file, just as you do for linear programming problems. Chapter 8, *More About Using ILOG CPLEX* in this manual describes file formats briefly, and the *ILOG CPLEX Reference Manual* documents file formats, such as MPS, LP, and others.

- To read MIP problem data into the Interactive Optimizer, use the `read` command with an option to indicate the file type.
- To read MIP problem data into your application, use the `importModel()` method in the Concert Technology Library or use a copy routine from the Callable Library. Table 5.1 summarizes the available routines and their purpose.

Table 5.1 *Callable Library Routines for Reading Formatted Files into MIP Applications*

To read this format	Use this Callable Library routine
MPS, LP, or SAV file	`CPXreadcopyprob()`
MST file containing MIP start values	`CPXreadcopymipstart()`
ORD file containing MIP priority order	`CPXreadcopyorder()`
ORD file containing MIP branching directions	`CPXreadcopyorder()`
SOS file	`CPXreadcopysos()`
MPS file containing SOS information	`CPXreadcopyprob()`

Displaying MIP Problems

If you are licensed to use the ILOG CPLEX Mixed Integer Optimizer, then you will see additional display options in the Interactive Optimizer. Table 5.2 summarizes these additional options.

Table 5.2 *Interactive Optimizer Display Options for MIP Problems*

Interactive command	Purpose
`display problem binaries`	lists variables restricted to binary values
`display problem generals`	lists variables restricted to integer values
`display problem semi-continuous`	lists variables of type semi-continuous and semi-integer
`display problem integers`	lists all of the above
`display problem sos`	lists the names of variables in one or more Special Ordered Sets
`display problem stats`	lists LP statistics plus: • binary variable types, if present; • general variable types, if present; • and number of SOS, if present.

In the Concert Technology Library, use one of the accessor methods supplied with the appropriate object class, such as `IloSOS2::getVariables`. Refer to the *ILOG Concert Technology Reference Manual* for more information.

From the Callable Library, use the routines `CPXgetctype()` and `CPXgetsos()` to access this information.

Determining Problem Type and Variable Type in MIPs

When you enter a problem in the Interactive Optimizer, ILOG CPLEX determines the problem type from the available information. If there are no binary variables, no general variables, and no SOS, ILOG CPLEX treats the problem type as LP. Before any variables can be changed to binary or general type (that is, restricted to integer values), the *problem type* must be changed to MIP. Reading an SOS description automatically changes the problem type to MIP.

Changing Problem Type

If you are licensed to use the ILOG CPLEX Mixed Integer Optimizer, then you will see additional `change` options in the Interactive Optimizer. To change the problem type to MIP, use the command `change problem mip.`

The command `change problem` shows you the type of the current problem and prompts you to indicate the type of problem you would like it to be. In other words, with this command, you can change the current MIP problem to its continuous relaxation or to its fixed MIP. Its continuous relaxation is a linear program in which all its variables are continuous (rather than restricted to integer values). Its fixed MIP is the linear program in which the integer variables are fixed at the values they attained in the best integer solution.

Since a continuous relaxation of a MIP and the fixed MIP are both linear programs, all the features of the ILOG CPLEX Interactive Optimizer are available to them, including information about the quality of solutions and about sensitivity analysis. The original variable bounds and their types are restored when the problem type is changed back to MIP.

Changing Variable Type

The command `change type` adds (or removes) the restriction on a variable that it must be an integer. In the Interactive Optimizer, when you enter the command `change type`, the system prompts you to enter the variable that you want to change, and then it prompts you to enter the type (`c` for continuous, `b` for binary, `i` for general integer, `s` for semi-continuous, `n` for semi-integer).

You can change a variable to binary even if its bounds are not *0* (zero) and *1* (one). However, in such a case, the system issues a warning message at optimization, and the optimization may terminate with a bound violation.

Consequently, in the example that we mentioned (see *Sample: Stating a MIP Problem* on page 152), if we want to make x_4 a binary variable, we should first change the *bounds* on x_4 to *0* and *1*; then we can safely change its type to *binary.*

If you change a variable's type to be semi-continuous or semi-integer, make sure to create both a lower bound and an upper bound for it. These variable types specify that at an optimal solution the value for the variable must be either exactly zero or else be between the lower and upper bounds (and further subject to the restriction that the value be an integer, in the case of semi-integer variables).

By the way, if its type has been changed to MIP, a problem may be a mixed integer problem, even if all its variables are continuous.

Modifying Relaxed and Fixed Problems

The `change` command in the Interactive Optimizer does not apply to the continuous relaxation of a MIP; nor does it apply to the fixed MIP. If you want to interactively modify and re-solve a relaxed or fixed version of a MIP, then you should follow these steps:

1. Write out the relaxed or fixed version to a file.
2. Read back in the relaxed or fixed version.

If you simply change the problem type to LP, all the MIP-related information will be discarded. After this modification, you will not be able to restore the original MIP problem.

Using the Mixed Integer Optimizer

The ILOG CPLEX Mixed Integer Optimizer exploits a *branch & cut algorithm*.

To invoke the Mixed Integer Optimizer:

◆ In the Interactive Optimizer, use the `mipopt` command.

◆ In the Concert Technology Library, with the method `IloCplex::solve().`

◆ In the Callable Library, you call it with the `CPXmipopt()` routine.

Branch & Cut

In the branch & cut algorithm, ILOG CPLEX solves a series of LP subproblems. To manage those subproblems efficiently, ILOG CPLEX builds a tree in which each subproblem is a node. The root of the tree is the *LP relaxation* of the original MIP problem.

If the solution to the relaxation has one or more fractional variables, ILOG CPLEX will try to find cuts. Cuts are constraints that cut away areas of the feasible region of the relaxation that contain fractional solutions. ILOG CPLEX can generate several types of cuts. (*Cuts* on page 159 tells you more about that topic.) Such algorithms have been known historically as branch & bound, especially when cuts are not generated.

If the solution to the relaxation still has one or more fractional-valued integer variables after ILOG CPLEX tries to add cuts, then ILOG CPLEX branches on a fractional variable to generate two new subproblems, each with more restrictive bounds on the branching variable. For example, with binary variables, one node will fix the variable at *0* (zero), the other, at *1* (one).

The subproblems may result in an all-integer solution, in an infeasible solution, or another fractional solution. If the solution is fractional, ILOG CPLEX repeats the process.

ILOG CPLEX cuts off nodes when the value of the objective function associated with the subproblem at that node is worse than the cutoff value. The cutoff value is determined in either of two ways:

- You set the cutoff value in the:
  - Interactive Optimizer by means of the command `set mip tolerances lowercutoff` (when you are maximizing the objective) or `set mip tolerances uppercutoff` (when you are minimizing the objective);
  - Concert Technology Library set the parameter `CutLo` or `CutUp`.
  - Callable Library set the parameter `CPX_PARAM_CUTLO` or `CPX_PARAM_CUTUP`.

  The default value of the lower cutoff is $-1e^{+75}$; the default value of the upper cutoff is $1e^{+75}$. You can supply any number that you find appropriate for your problem.
- ILOG CPLEX will use the value of the best integer solution found so far, as modified by tolerance parameters.
  - In the Interactive Optimizer, use the command `set mip tolerances objdifference` for an absolute objective difference cutoff or `set mip tolerances relobjdifference` for a relative objective difference cutoff.
  - In the Concert Technology Library set the parameter `ObjDif` or `RelObjDif`.
  - In the Callable Library, set the parameter `CPX_PARAM_OBJDIF or CPX_PARAM_RELOBJDIF`.

  Be careful in changing these tolerances: if either of them is nonzero, you may miss the optimal solution by as much as that amount. For example, if the true optimum is *100*, and the absolute cutoff is set to *5*, and a feasible solution of, say, *103* is found at some point, then the cutoff will discard all nodes with a solution worse than *98*, and thus the solution of *100* will be overlooked.

Once ILOG CPLEX finds an integer solution, it does the following:

- it makes that integer solution the *incumbent solution* and that node the *incumbent node*;
- it makes the value of the objective function at that node (modified by the objective difference parameter) the new cutoff value;
- it prunes from the tree all subproblems for which the value of the objective function is no better than the incumbent.

You control the path that ILOG CPLEX traverses in the tree through several parameters, as summarized in Table 5.3 and explained further in later sections. Briefly, ILOG CPLEX must make different kinds of choices:

- within a tree, about which node to branch on,
- at a node, which variable to branch on, and
- at a variable, which direction to branch (up or down or other).

Table 5.3 *Parameters for Controlling Branch & Cut Strategy*

Interactive Optimizer Command	Concert Technology Library Function	Callable Library Routine
`set mip strategy backtrack`	`IloCplex::setParam(BtTol, n)`	`CPXsetdblparam(env, CPX_PARAM_BTTOL, n)`
`set mip strategy nodeselect`	`IloCplex::setParam(NodeSel, i)`	`CPXsetintparam(env, CPX_PARAM_NODESEL, i)`
`set mip strategy variableselect`	`IloCplex::setParam(VarSel, i)`	`CPXsetintparam(env, CPX_PARAM_VARSEL, i)`
`set mip strategy bbinterval`	`IloCplex::setParam(BBInterval, i)`	`CPXsetintparam(env, CPX_PARAM_BBINTERVAL, i)`
`set mip strategy branch`	`IloCplex::setParam(BrDir, i)`	`CPXsetintparam(env, CPX_PARAM_BRDIR, i)`

At each node, ILOG CPLEX may delve deeper into the tree or it may backtrack. The value of the *backtrack parameter*, page 176, influences this decision. When ILOG CPLEX backtracks, there are usually large numbers of available, unexplored nodes. The *node selection parameter*, page 177, influences its selection. Once a node has been selected, the *variable selection parameter*, page 176, influences which variable is selected for branching. *Priority*, page 162, provides a powerful mechanism through which you supply problem-specific directives about the order of variables during branching. You also supply problem-specific preferences about *branching direction*, either globally or by specific variable. And special ordered sets (SOS), page 169, may also improve branching strategy.

Feasibility and Optimality

The parameter `IloCplex::MipEmphasis` / `CPX_PARAM_MIPEMPHASIS` (`set mip emphasis` in the Interactive Optimizer) specifies whether CPLEX should emphasize feasibility or optimality as it solves the problem.

At the default setting of 0, CPLEX uses tactics designed to find a proved optimal solution most quickly, with less regard for the speed at which feasible solutions are produced along the way. With a setting of 1, CPLEX uses tactics designed to find the first and subsequent feasible solutions more quickly, at the likely expense of prolonging the time required to find a proven optimal solution.

Either setting will deliver a proved optimum, will produce feasible solutions during the course of computation, and will honor other parameter settings (such as time limits or branching strategies); the difference is in the trade-offs the MIP algorithm makes between the competing aims. Since proving optimality is often far more difficult than finding feasible solutions, setting this parameter to 1 is useful in situations (for example) where obtaining a good solution within a time limit is more important than arriving at a proved optimum.

Cuts

Cuts are constraints added to a model to restrict (cut away) noninteger solutions that would otherwise be solutions of the LP relaxation. The addition of cuts usually reduces the number of branches needed to solve a MIP.

In the following descriptions of cuts, the term *subproblem* includes the root node (that is, the LP relaxation). Cuts are most frequently seen at the root node, but they may be added by ILOG CPLEX at other nodes as conditions warrant.

ILOG CPLEX generates its cuts in such a way that they are valid for all subproblems, even when they are discovered during analysis of a particular subproblem. If the solution to a subproblem violates one of the subsequent cuts, ILOG CPLEX may add an LP constraint to reflect this condition.

Clique Cuts

A *clique* is a relationship among a group of binary variables such that at most one variable in the group can be positive in any integer feasible solution. Before optimization starts, ILOG CPLEX constructs a graph representing these relationships and finds maximal cliques in the graph.

Cover Cuts

If a constraint takes the form of a *knapsack constraint* (that is, a sum of binary variables with nonnegative coefficients less than or equal to a nonnegative right-hand side), then there is a minimal cover associated with the constraint. A *minimal cover* is a subset of the variables of the inequality such that if all the subset variables were set to one, the knapsack constraint would be violated, but if any one subset variable were excluded, the constraint would be satisfied. ILOG CPLEX can generate a constraint corresponding to this condition, and this cut is called a *cover cut*.

Disjunctive Cuts

A MIP problem can be divided into two subproblems with disjunctive feasible regions of their LP relaxations by branching on an integer variable. *Disjunctive cuts* are inequalities valid for the feasible regions of LP relaxations of the subproblems, but not valid for the feasible region of LP relaxation of the MIP problem.

Flow Cover Cuts

Flow covers are generated from constraints that contain continuous variables, where the continuous variables have variable upper bounds that are zero or positive depending on the setting of associated binary variables. The idea of a flow cover comes from considering the constraint containing the continuous variables as describing a single node in a network where the continuous variables are in-flows and out-flows. The flows will be on or off depending on the settings of the associated binary variables for the variable upper bounds. The flows and the demand at the single node imply a knapsack constraint. That knapsack constraint is then used to generate a cover cut on the flows (that is, on the continuous variables and their variable upper bounds).

Flow Path Cuts

Flow path cuts are generated by considering a set of constraints containing the continuous variables that describe a path structure in a network, where the constraints are nodes and the continuous variables are in-flows and out-flows. The flows will be on or off depending on the settings of the associated binary variables.

Gomory Fractional Cuts

Gomory fractional cuts are generated by applying integer rounding on a pivot row in the optimal LP tableau for a (basic) integer variable with a fractional solution value.

Generalized Upper Bound (GUB) Cover Cuts

A *GUB constraint* for a set of binary variables is a sum of variables less than or equal to one. If the variables in a GUB constraint are also members of a knapsack constraint, then the minimal cover can be selected with the additional consideration that at most one of the members of the GUB constraint can be one in a solution. This additional restriction makes the *GUB cover cuts* stronger (that is, more restrictive) than ordinary cover cuts.

Implied Bound Cuts

In some models, binary variables imply bounds on continuous variables. ILOG CPLEX generates potential cuts to reflect these relationships.

Mixed Integer Rounding (MIR) Cuts

MIR cuts are generated by applying integer rounding on the coefficients of integer variables and the right-hand side of a constraint.

Adding Cuts and Re-Optimizing

Each time ILOG CPLEX adds a cut, the subproblem is re-optimized. CPLEX repeats the process of adding cuts at a node until it finds no further effective cuts. It then selects the branching variable for the subproblem.

Parameters control the way each class of cuts is used. Those parameters are listed in Table 5.4.

Table 5.4 *Parameters for Controlling Cuts*

Cut Type	Interactive Command	Concert Technology Library Parameter	Callable Library Parameter
Clique	`set mip cuts cliques`	`IloCplex::Cliques`	`CPX_PARAM_CLIQUES`
Cover	`set mip cuts covers`	`IloCplex::Covers`	`CPX_PARAM_COVERS`
Disjunctive	`set mip cuts disjunctive`	`IloCplex::DisjCuts`	`CPX_PARAM_DISJCUTS`

Table 5.4 *Parameters for Controlling Cuts (Continued)*

Cut Type	**Interactive Command**	**Concert Technology Library Parameter**	**Callable Library Parameter**
Flow Cover	`set mip cuts flowcuts`	`IloCplex::FlowCovers`	`CPX_PARAM_FLOWCOVERS`
Flow Path	`set mip cuts pathcut`	`IloCplex::FlowPaths`	`CPX_PARAM_FLOWPATHS`
Gomory	`set mip cuts gomory`	`IloCplex::FracCuts`	`CPX_PARAM_FRACCUTS`
GUB Cover	`set mip cuts gubcovers`	`IloCplex::GUBCovers`	`CPX_PARAM_GUBCOVERS`
Implied Bound	`set mip cuts implied`	`IloCplex::ImplBd`	`CPX_PARAM_IMPLBD`
Mixed Integer Rounding (MIR)	`set mip cuts mircut`	`IloCplex::MIRCuts`	`CPX_PARAM_MIRCUTS`

The default value of each of those parameters is `0` (zero). By default, ILOG CPLEX automatically determines how often (if at all) it should try to generate that class of cut. A setting of `-1` indicates that no cuts of the class should be generated; a setting of `1` indicates that cuts of the class should be generated moderately; and a setting of `2` indicates that cuts of the class should be generated aggressively. For disjunctive cuts, a setting of 3 is permitted, which indicates that disjunctive cuts should be generated very aggressively.

In the Interactive Optimizer, the command `set mip cuts all` ***i*** applies the value *`i`* to all classes of cut parameters. That is, you can set them all at once.

The cuts-factor parameter controls the number of cuts ILOG CPLEX adds to the model. The problem can grow to cuts-factor times the original number of rows in the model (or in the presolved model, if the presolver is active). Thus, a cuts-factor of 1.0 would mean that no cuts will be generated, which may be a more convenient way of turning off all cuts than setting them individually. The default cuts-factor value of 4.0 works well in most cases, as it allows a generous number of cuts while in rare instances it also serves to limit unchecked growth in the problem size.

Set the cuts-factor parameter in the:

◆ Interactive Optimizer with the command `set mip limits cutsfactor`.

◆ Concert Technology Library with the function `IloCplex::setParam(CutsFactor,` *`n`*`).`

◆ Callable Library with the routine `CPXsetdblparam(env, CPX_PARAM_CUTSFACTOR,` *`n`*).

The cuts aggregation parameter controls the number of constraints allowed to be aggregated for generating MIR and flow cover cuts. Set this parameter in the:

- ◆ Interactive Optimizer with the command `set mip limits aggforcut`.
- ◆ Concert Technology Library with the function `IloCplex::setParam(AggCutLim, i)`.
- ◆ Callable Library with the routine `CPXsetintparam(env, CPX_PARAM_AGGCUTLIM, i)`.

The gomorypass parameter controls the number of passes for generating Gomory fractional cuts. Set this parameter in the:

- ◆ Interactive Optimizer with the command `set mip limits gomorypass`.
- ◆ Concert Technology Library with the function `IloCplex::setParam(FracPass, i)`.
- ◆ Callable Library with the routine `CPXsetintparam(env, CPX_PARAM_FRACPASS, i)`.

The parameter will not have any effect if the parameter for `set mip cuts gomory` has a nonzero value. The gomorycand parameter controls the number of variable candidates to be considered for generating Gomory fractional cuts. Set this parameter in the:

- ◆ Interactive Optimizer with the command `set mip limits gomorycand`.
- ◆ Concert Technology Library with the function `IloCplex::setParam(FracCand, i)`.
- ◆ Callable Library with the routine `CPXsetintparam(env, CPX_PARAM_FRACCAND, i)`.

Priority

In branch & cut, ILOG CPLEX makes decisions about which variable to branch on at a node. You can control the order in which ILOG CPLEX branches on variables by issuing a *priority order.* A priority order assigns a branching priority to some or all of the integer variables in a model. ILOG CPLEX branches on variables with an assigned priority before variables without a priority. It also branches on variables with higher priority before variables with lower priority, when the variables have fractional values.

You can specify priority for any variable, though the priority is used only if the variable is a general integer variable, a binary integer variable, or a member of a special ordered set.

Sometimes, a generic priority may be helpful. There are options for setting priority among variables based on the magnitude of their coefficients in the objective function, on the range of their bounds, and on their objective value divided by column count.

For example:

- ◆ In the Interactive Optimizer, the command `set mip ordertype 1` will make ILOG CPLEX branch on variables by decreasing cost.
- ◆ For the Concert Technology Library, the corresponding parameter is `IloCplex::MIPOrdType`.
- ◆ For the Callable Library it is `CPX_PARAM_MIPORDTYPE.`

If you explicitly read a file of priority orders, its settings will override any generic priority order you may have set by interactive commands.

- ◆ In the Interactive Optimizer, the command `set mip strategy order 0` overrides all priority orders—whether set by a command or from a file—so that ILOG CPLEX uses no priority orders.
- ◆ For the Concert Technology Library the corresponding parameter is `IloCplex::MIPOrdInt`.
- ◆ For the Callable Library it is `CPX_PARAM_MIPORDIND.`

Problems that use integer variables to represent different types of decisions should assign higher priority to those that must be decided first. For example, if some variables in a model activate processes, and others use those activated processes, then the first group of variables should be assigned higher priority than the second group. In that way, you can use priority to achieve better solutions.

Priority based on the magnitude of objective coefficients is often useful in this way.

Heuristics

CPLEX supports a heuristic to find integer solutions at nodes during the branch & cut procedure. To invoke this heuristic:

- ◆ In the Interactive Optimizer, use the command `set mip strategy heuristicfreq`.
- ◆ In the Concert Technology Library, set the parameter `IloCplex::HeurFreq`.
- ◆ From the Callable Library, set the parameter `CPX_PARAM_HEURFREQ`.

For example, if the frequency is set to `20`, then the node heuristic will be applied at node 0, node 20, node 40, and so on. At the default setting `0` (zero), ILOG CPLEX automatically determines the frequency dynamically. The value `-1` turns this feature off.

Preprocessing: Presolver and Aggregator

When you invoke the MIP optimizer—whether through the Interactive Optimizer command `mipopt`, through a call to the Concert Technology Library function `IloCplex::solve()`, or through the Callable Library routine `CPXmipopt()`—ILOG CPLEX by default automatically preprocesses your problem. Table 5.5 summarizes the preprocessing parameters. In preprocessing, ILOG CPLEX applies its presolver and aggregator once or more to reduce the size of the integer program in order to strengthen the initial linear relaxation and to decrease the overall size of the mixed integer program.

Table 5.5 *Parameters for Controlling MIP Preprocessing*

Interactive Command	**Concert Technology Library Parameter**	**Callable Library Parameter**	**Comment**
`set preprocessing aggregator`	`IloCplex::AggInd`	`CPX_PARAM_AGGIND`	on by default
`set preprocessing presolve`	`IloCplex::PreInd`	`CPX_PARAM_PREIND`	on by default
`set preprocessing boundstrength`	`IloCplex::BndStrenInd`	`CPX_PARAM_BNDSTRENIND`	presolve must be on
`set preprocessing coeffreduce`	`IloCplex::CoeRedInd`	`CPX_PARAM_COEREDIND`	presolve must be on
`set preprocessing relax`	`IloCplex::RelaxPreInd`	`CPX_PARAM_RELAXPREIND`	applies to relaxation
`set preprocessing reduce`	`IloCplex::Reduce`	`CPX_PARAM_REDUCE`	all on by default
`set preprocessing numpass`	not available	`CPX_PARAM_PREPASS`	automatic by default

The parameters reduce and numpass have the same meanings for LP and MIP. *Preprocessing: Presolver and Aggregator* on page 98 explains the meanings and adjustments of all these parameters.

While preprocessing, ILOG CPLEX also attempts to strengthen bounds on variables. This bound strengthening may take a long time. In such cases, you may want to turn off bound strengthening.

ILOG CPLEX also attempts to reduce coefficients during preprocessing. Coefficient reduction usually strengthens the linear programming relaxation and reduces the number of nodes in the branch & cut tree, but not always. Sometimes, it increases the amount of time needed to solve the linear programs at each node—enough time to offset the benefit of fewer nodes. Two levels of coefficient reduction are available, so it is worthwhile to experiment with these preprocessing options to see whether they are beneficial to your problem.

In addition, you may also set the relaxation parameter to tell ILOG CPLEX to apply preprocessing to the initial relaxation of the problem. Sometimes this preprocessing will result in additional, beneficial presolve transformations in the LP relaxation—transformations that are not possible in the original MIP model.

ILOG CPLEX preprocesses a MIP by default. However, if you use a basis to start LP optimization at the root node, ILOG CPLEX will proceed with that starting basis without preprocessing it. In other words, if you change a MIP to a relaxed problem, optimize it as an LP, and use that basis to start MIP-optimization, then no preprocessing will occur.

If you want to apply a particular LP algorithm to the first relaxation, this strategy is reasonable. However, for problems that benefit from MIP preprocessing, we do *not* recommend it. Instead, we recommend that you use parameters to indicate which algorithm to use on the first relaxation (`startalgorithm` in the Interactive Optimizer and `CPX_PARAM_STARTALG` in the Callable Library) and which to use on the subproblems (`subalgorithm` in the Interactive Optimizer and `CPX_PARAM_SUBALG` in the Callable Library). In Concert Technology Library, use the methods `IloCplex::setRootAlgorithm` and `IloCplex::setNodeAlgorithm`. *Subproblem Optimization* on page 187 explains more about choosing algorithms for the first relaxation and subsequent subproblems.

Starting from a Solution

You can provide a known solution (for example, from a MIP problem previously solved or from your knowledge of the problem) to serve as the first integer solution. In such a start, you must specify values for all integer variables, for all semi-continuous variables, and for all members of special ordered sets. Optionally, you may also specify values for continuous variables. ILOG CPLEX evaluates that start solution for integrality and feasibility. If it is integer-feasible, it will become an integer solution of the current problem.

Occasionally, a set of MIP start values will be integer feasible for the original problem, but not feasible for the preprocessed problem because of complicated transformations carried out by the presolver or aggregator. ILOG CPLEX issues a warning whenever the MIP start values do not provide an integer solution, and optimization continues.

You control whether ILOG CPLEX uses a MIP start solution through the mipstart parameter.

- In the Interactive Optimizer, use the command `set mip strategy mipstart 1`.
- For the Concert Technology Library, use the method `IloCplex::setParam(MIPStart, IloTrue)`.
- For the Callable Library, use the routine `CPXsetintparam(env, CPX_PARAM_MIPSTART, CPX_ON)`.

ILOG CPLEX reads and writes MIP start information in MST files (that is, MIP start-file format, as described briefly in*Understanding File Formats* on page 264 or documented in the *ILOG CPLEX Reference Manual*).

CPLEX saves starting values for all integer variables, all semi-continuous variables, and all members of special ordered sets at the end of MIP optimization when there is a feasible solution. These values can then be used in subsequent optimizations.

- In the Interactive Optimizer, use the `write` command to generate an MST file.
- In the Concert Technology Library, use the method `IloCplex::exportModel()`.

- In the Callable Library, use the routine `CPXmstwrite()`.

Termination

ILOG CPLEX terminates MIP optimization under a variety of circumstances. First, ILOG CPLEX declares integer optimality and terminates when it finds an integer solution and all nodes have been processed. Optimality in this case is relative to whatever tolerances and optimality criteria you have set. For example, ILOG CPLEX considers the cutoff value and the objective difference parameter in this context.

In addition, ILOG CPLEX terminates optimization when it reaches a limit that you have set. You can set limits on time, number of nodes, size of tree memory, size of the node log file, and number of integer solutions. Table 5.6 summarizes those parameters and their purpose.

***Table 5.6** Parameters to limit MIP optimization*

To set a limit on	Use this parameter		
	Concert Technology Library	**Callable Library**	**Interactive Optimizer**
elapsed time	`IloCplex::TiLim`	`CPX_PARAM_TILIM`	`timelimit`
number of nodes	`IloCplex::NodeLim`	`CPX_PARAM_NODELIM`	`mip limits nodes`
size of tree memory	`IloCplex::TreLim`	`CPX_PARAM_TRELIM`	`mip limits treememory`
size of node log file	`IloCplex::WorkMem`	`CPX_PARAM_WORKMEM`	`workmem`
number of integer solutions	`IloCplex::IntSolLim`	`CPX_PARAM_INTSOLLIM`	`mip limits solutions`

The limit on tree memory terminates optimization only when the parameter controlling the node file (in the Interactive Optimizer, `mip strategy file`, in the Concert Technology Library, `IloCplex::NodeFileInd`, in the Callable Library, `CPX_PARAM_NODEFILEIND)` is `0`, the default. If the value is other than `0`, optimization will continue.

ILOG CPLEX also terminates when an error occurs, such as when ILOG CPLEX runs out of memory or when a subproblem cannot be solved. If an error is due to failure to solve a subproblem, an additional line appears in the node log file to indicate the reason for that failure.

Writing a Tree File

When ILOG CPLEX terminates a MIP optimization before it achieves optimality (for example, because it has reached a limit you set), it still has significant information about the current branch & cut tree. You can save this information by writing it to a file of type TRE (a binary, proprietary ILOG CPLEX format). Later, you can then read the saved TRE file and restart the optimization from where ILOG CPLEX left off.

To save a MIP in a TRE file:

- In the Interactive Optimizer, use the command `write filename.tre` or `write filename tre`.
- From the Callable Library, use the routine `CPXtreewrite()`.

A TRE file may be quite large (corresponding to the current size of an active tree) so it may consume considerable disk space.

If you modify the model of a MIP *after* you create its TRE file, then the TRE file will be of no use to you. ILOG CPLEX will accept the old TRE file if the basic dimensions of the problem have not changed, but the results it produces from it will likely be invalid for the modified model.

Post-Solution Information in a MIP

Interpreting Solution Statistics on page 114 explains how to use the `display` command in the Interactive Optimizer to see post-solution information from the *linear* optimizers. However, because of the way integer solutions are generated, the `display` command shows you only limited information from the MIP optimizer. In fact, ILOG CPLEX generates integer solutions by solving subproblems that have different bounds from the original problem, so computing solution values with the original bounds will not usually give the same solution. Nevertheless, the following solution statistics are available from the MIP optimizer:

- objective function value for the best integer solution, if one exists;
- best bound, that is, best objective function value among remaining subproblems;
- solution quality;
- primal values for the best integer solution, if one has been found;
- slack values for best integer solution, if one has been found.

If you request other solution statistics, ILOG CPLEX will issue the error message, "Not available for mixed integer problems—use CHANGE PROBLEM to change the problem type."

Using Sensitivity Information in a MIP

Other post-solution information does not have the same meaning in a mixed integer program as in a linear program because of the special nature of the integer variables in the MIP. The reduced costs, dual values, and sensitivity ranges give you information about the effect of making small changes in problem data so long as *feasibility is maintained*. Integer variables, however, *lose feasibility* if a small change is made in their value, so this post-solution information cannot be used to evaluate changes in problem data in the usual way of LPs.

Integer variables typically represent major structural decisions in a model, and often many continuous variables of the model are related to these major decisions. With that observation in mind, if you take the integer variable values as given, then you can use post-solution information *applying only to the continuous variables* in the usual way.

To access this limited sensitivity information in a MIP:

- In the Interactive Optimizer, use the command `change problem fixed` to fix the values of the integer variables.
- In the Callable Library, use the routine `CPXchgprobtype()`.

ILOG CPLEX then sets the variable bounds so that upper and lower bounds are those in the current integer solution. You can then optimize the resulting linear program and display its post-solution statistics.

Using Special Ordered Sets (SOS)

A special ordered set (SOS) is an additional way to specify integrality conditions in a model. There are various types of SOS:

- SOS Type 1 is a set of variables (whether all integer, all continuous, or mixed integer and continuous) where at most one variable may be nonzero.
- SOS Type 2 is a set or integer or continuous variables where at most two variables may be nonzero. If two variables are nonzero, they must be adjacent in the set.

ILOG CPLEX uses special branching strategies to take advantage of SOS. The special branching strategies depend upon the order among the variables in the set. The order is specified by assigning weights to each variable. The order of the variables in the model (such as in the MPS or LP format data file, or the column index in a Callable Library application) is ***not*** used in SOS branching. If there is no order relationship among the variables (such that weights cannot be specified or would not be meaningful), SOS branching should not be used. For many classes of problems, these branching strategies can significantly improve performance.

Example: SOS Type 1 for Sizing a Warehouse

To give you a feel for how SOS can be useful, here's an example of an SOS Type 1 used to choose the size of a warehouse. Let's assume for this example that we can build a warehouse of *10000*, *20000*, *40000*, or *50000* square feet. We define binary variables for the four sizes, say, x_1, x_2, x_4, and x_5. We connect these variables by a constraint defining another variable to denote available square feet, like this: $z - 10000x_1 - 20000x_2 - 40000x_4 - 50000x_5 = 0$.

Those four variables are members of a special ordered set. Only one size can be chosen for the warehouse; that is, at most one of the *x* variables can be nonzero in the solution. And, there is an order relationship among the *x* variables (namely, the sizes) that can be used as weights. We say that the *weights* of the set members are *10000*, *20000*, *40000*, and *50000*.

Let's say furthermore that we have a fractional (that is, noninteger) solution of $x_1 = 0.1$, $x_5 = 0.9$. These values indicate that other parts of the model have imposed the requirement of *46000* square feet since $0.1*10000 + 0.9*50000 = 46000$. In SOS parlance, we say that the *weighted average* of the set is $(0.1*10000 + 0.9*50000)/(0.1 + 0.9) = 46000$.

We *split* the set before the variable with weight exceeding the weighted average. In this case, we split the set like this: x_1, x_2, and x_4 will be in one subset; x_5 in the other.

Now we branch. One branch restricts x_1, x_2, x_4 to *0* (zero). This branch results in x_5 being set to *1* (one).

The other branch, where x_5 is set to *0* (zero), results in an infeasible solution, so we remove it from further consideration.

If a warehouse must be built, then we need the additional constraint that $x_1 + x_2 + x_4 + x_5 = 1$. The implicit constraint for an SOS Type 1 is less than or equal to one. The linear programming relaxation may more closely resemble the MIP if we add that constraint.

Declaring SOS Members

ILOG CPLEX offers you several ways to declare an SOS in a problem:

◆ Use an SOS file (that is, one in SOS format, with the file extension `.sos`). SOS files offer you the most powerful and flexible alternative because the SOS file structure allows you to do several tasks at once:

- provide branching priorities for sets,
- assign weights to individual set members,
- define overlapping sets.

◆ Use SOS declarations within an MPS or LP file (that is, one in MPS format with the file extension `.mps` or in LP format with the file extension `.lp`. If you already have MPS files with SOS information, you may prefer this option. Conventions for declaring SOS information in MPS files are documented in the *ILOG CPLEX Reference Manual*.

Setting Branching Priority for an SOS

An entire SOS can be given a branching priority. There are two alternative ways to give an SOS branching priority, both documented in the *ILOG CPLEX Reference Manual*:

◆ Use an SOS file to set priorities.

- Use an ORD file to set priorities.

ILOG CPLEX derives the branching priority of a *set* from the branching priorities of its *members*: the entire set is assigned the highest priority among its members.

To specify SOS priorities:

- In the Concert Technology Library, use the functions `IloCplex::setPriority()` and `IloCplex::setPriorities()`.
- In the Callable Library, use the routines `CPXcopysos()` or `CPXcopyorder()`.

Assigning SOS Weights

Members of an SOS should be given unique weights that in turn define the order of the variables in the set. (These unique weights are also called *reference row values*.) The most flexible way for you to assign weights is through an SOS, MPS, or LP file. An alternative is to use MPS format to assign a single reference row containing weights. Such a reference row may be a free row with specific weighting information, or it may be the objective function, or it may be a constraint row.

- In the Concert Technology library, SOS weights are specified in the constructor when the SOS is created.
- In the Callable Library, the routine `CPXcopysos()` lets you specify weights directly in an application.

In our SOS example, page 168, we used the coefficients of the warehouse capacity constraint to assign weights.

Using Semi-Continuous Variables

Semi-continuous variables are variables that may take either the value 0 or values in a finite range [a, b]. Semi-continuous variables can be specified in MPS and LP files. In the Concert Technology Library, semi-continuous variables are instances of the class `IloSemiContVar`. In the Callable Library, semi-continuous variables can be entered with type `CPX_SEMICONT` or `CPX_SEMIINT` via the routine `CPXcopycttype()`.

Progress Reports: Interpreting the Node Log

As we explained earlier, when ILOG CPLEX optimizes mixed integer programs, it builds a tree with the linear relaxation of the original MIP at the root and subproblems to optimize at the nodes of the tree. ILOG CPLEX reports its progress in optimizing the original problem in a *node log file* as it traverses this tree.

Through ILOG CPLEX parameters, you control how information in the log file is recorded and displayed. You can use those parameters at their default values (adequate for most problems), or you can reset them through commands in the Interactive Optimizer, through member functions in the Concert Technology Library, or through routines from the Callable Library. Table 5.7 summarizes those parameters, and the following paragraphs explain how to use them.

Table 5.7 *Parameters for Controlling the ILOG CPLEX Node Log File*

Default	Interactive Command	Concert Technology Library Function	Callable Library Routine
2	`set mip display`	`IloCplex::setParam(MIPDisplay, i)`	`CPXsetintparam(env,CPX_PARAM_MIPDISPLAY,i)`
100	`set mip interval`	`IloCplex::setParam(MIPInterval, i)`	`CPXsetintparam(env,CPX_PARAM_MIPINTERVAL,i)`

Generally, ILOG CPLEX records a line in the node log about every node with an integer solution and about every *n* nodes solved, where *n* is controlled by the MIP interval parameter.

◆ In the Interactive Optimizer, use the command `set mip interval` *`i`* to change the MIP interval parameter in order to log node information more (a smaller value of *`i`*) or less (a larger value of *`i`*) frequently.

◆ From the Callable Library, use the routine `CPXsetintparam()` with arguments to indicate the environment, the parameter `CPX_PARAM_MIPINTERVAL`, and a positive integer value. The default value is `100`.

Here is an example of such a node log file:

```
Tried aggregator 1 time.
No MIP presolve or aggregator reductions.
Presolve time =    0.00 sec.
Root relaxation solution time = 0.00 sec
Objective is integral.

     Nodes                                     Cuts/
  Node Left  Objective  IInf  Best Integer   Best Node  ItCnt Gap

     0   0     4.0000     6                    4.0000     12
*    4   2     5.0000     0        5.0000      4.0000     17 20.00%
    10   1     cutoff              5.0000      4.0000     31 20.00%

Integer optimal solution:  Objective =   5.0000000000e+000
Solution time =    0.02 sec.    Iterations = 41   Nodes = 13
```

In that example, ILOG CPLEX found the optimal objective function value of 5 at 13 of the nodes in 41 iterations, and ILOG CPLEX found an optimal integer solution at node 4. The MIP interval parameter was set at 10, so every tenth node was logged, in addition to the node where an integer solution was found.

As you can see in that example, ILOG CPLEX logs an asterisk (`*`) in the left-most column for any node where it finds an integer-feasible solution. In the next column, it logs the *node number*. It next logs the number of nodes left to explore.

In the next column, ILOG CPLEX either records the *objective value* at the node or a *reason to fathom* the node. (A node is fathomed if the solution of a subproblem at the node is infeasible; or if the value of objective function at the node is worse than the cutoff value for branch & cut; or if the node supplies an integer solution.)

In the column labeled `IInf`, ILOG CPLEX records the number of integer-infeasible variables and special ordered sets. If no solution has been found, the next column is left blank; otherwise, it records the best integer solution found so far.

The column labeled `Cuts/Best Node` records the best objective function value of all the unexplored nodes. If the word `Cuts` appears in this column, it means various cuts were generated; if a particular name of a cut appears, then only that kind of cut was generated.

The column labeled `ItCnt` records the cumulative iteration count of the algorithm solving the subproblems. Until a solution has been found, the column labeled `Gap` is blank. If a solution has been found, the relative gap value is printed when it is less than `999.99`; otherwise, hyphens are printed. The gap is computed as `abs(best integer - best node)/(1e-10 + abs(best integer))`. Consequently, the printed gap value may not always move smoothly. In particular, there may be sharp improvements whenever a new best integer solution is found.

ILOG CPLEX also logs its addition of cuts to a model. Here is an example of a node log file from a problem where ILOG CPLEX made cover cuts.

```
MIP Presolve eliminated 0 rows and 1 columns.
MIP Presolve modified 12 coefficients.
Reduced MIP has 15 rows, 32 columns, and 97 nonzeros.
Presolve time =    0.00 sec.

    Nodes                                        Cuts/
 Node  Left     Objective  IInf  Best Integer  Best Node   ItCnt Gap

    0     0    2819.3574     7                 2819.3574     35
               2881.8340     8                 Covers: 4     44
               2881.8340    12                 Covers: 3     48
*   7     6    3089.0000     0     3089.0000   2904.0815     62 5.99%

Cover cuts applied:  30

Integer optimal solution:  Objective =   3.0890000000e+003
Solution time =    0.10 sec.    Iterations = 192   Nodes = 44
```

ILOG CPLEX also logs the number of clique inequalities in the clique table at the beginning of optimization and the number eventually applied. Cuts generated at intermediate nodes are not logged individually unless they happen to be generated at a node logged for other reasons. ILOG CPLEX logs the number of applied cuts of all classes at the end.

CPLEX also indicates, in the node log file, each instance of a successful application of the node heuristic. The following example shows a node log file for a problem where the heuristic found a solution at node 0. The + denotes a node generated by the heuristic.

```
       Nodes                                         Cuts/
  Node  Left   Objective  IInf  Best Integer  Best Node  ItCnt Gap
    0     0      403.8465  640                 403.8465   4037
                 405.2839  609              Cliques: 10   5208
                 405.2891  612              Cliques:  2   5288
Heuristic: feasible at 437.000, still looking
Heuristic: feasible at 437.000, still looking
Heuristic complete
*   0+    0      436.0000    0      436.0000   405.2891   5288 7.04%
```

Periodically, if the MIP display parameter is greater than `0` (zero), ILOG CPLEX records the cumulative time spent since the beginning of the current MIP optimization and the amount of memory used by branch & cut. (By periodically, we mean that time and memory information appears either every `20` nodes or ten times the MIP display parameter, whichever is greater. The default value of the MIP display parameter is `2`.) The following example shows you one line from a node log file indicating elapsed time and memory use.

```
Elapsed b&b time = 120.01 sec. (tree size =  0.09 MB)
```

To change the MIP display parameter:

- ◆ In the Interactive Optimizer, use the command `set mip display`.
- ◆ From the Callable Library, use the routine `CPXsetintparam()` with arguments to indicate the environment, the parameter `CPX_PARAM_MIPDISPLAY`, and a value.

Table 5.8 lists the acceptable values for this parameter.

Table 5.8 *Values of the MIP Display Parameter*

Value	Effect
`0`	no display
`1`	display integer feasible solutions
`2`	display nodes under mip interval control
`3`	same as 2, but add information on node cuts
`4`	same as 3, but add LP display for root node
`5`	same as 3, but add LP display for all nodes

ILOG CPLEX prints an additional summary line in the log if optimization stops before it is complete. This summary line shows the best MIP bound, that is, the best objective value among all the remaining node subproblems. The following example shows you lines from a node log file where an integer solution has not yet been found, and the best remaining objective value is 2973.9912281.

```
Node limit, no integer solution.
Current MIP best bound =    2.9739912281e+03 (gap is infinite)
Solution time =    0.01 sec.  Iterations = 68  Nodes = 7 (7)
```

Sample: Stating a MIP Problem on page 152 offers a typical MIP problem. Here is the node log file for that problem with the default setting of the MIP display parameter:

```
Tried aggregator 1 time.
Aggregator did 1 substitutions.
Reduced MIP has 2 rows, 3 columns, and 6 nonzeros.
Presolve time =    0.00 sec.
Clique table:0 GUB, 0 GUBEQ, 0 two-covers, 0 probed
ImplBd table: 0 bounds
Root relaxation solution time =    0.00 sec.

      Nodes                                    Cuts/
 Node  Left  Objective  IInf  Best Integer  Best Node  ItCnt Gap

   0     0    125.2083     1                125.2083      3
*             122.5000     0      122.5000   Cuts:  2      4

Mixed integer rounding cuts applied: 1
Gomory fractoinal cuts applied: 1

Integer optimal solution:  Objective =   1.2250000000e+002
Solution time =    0.02 sec.    Iterations = 4   Nodes = 0
```

These additional items appear only in the node log file (not on screen):

◆ `Variable` records the name of the variable where ILOG CPLEX branched to create this node. If the branch was due to a special ordered set, the name listed here will be the right-most variable in the left subset.

◆ `B` indicates the branching direction:

- `D` means the variables was restricted to a lower value;
- `U` means the variable was restricted to a higher value;
- `L` means the left subset of the special ordered set was restricted to 0 (zero);
- `R` means the right subset of the special ordered set was restricted to 0 (zero).

◆ `Parent` indicates the node number of the parent.

◆ `Depth` indicates the depth of this node in the branch & cut tree.

Troubleshooting MIP Performance Problems

Even the most sophisticated methods currently available to solve pure integer and mixed integer programming problems require noticeably more computation than the methods for similarly sized pure linear programs. Many relatively small integer programming models, in fact, still take enormous amounts of computing time to solve. Indeed, some such models have never yet been solved. In the face of these practical obstacles to a solution, proper formulation of the model is crucial to successful solution of pure integer or mixed integer programs.

For help in formulating a model of your own integer or mixed integer problem, you may want to consult H.P. Williams's textbook about practical model building (referenced in *Further Reading* on page 25 in this manual).

Also you may want to develop a better understanding of branch & cut, a feature of the ILOG CPLEX MIP Optimizer. For that purpose, Williams's book offers a good introduction, and Nemhauser and Wolsey's book (also referenced in *Further Reading* on page 25 in this manual) goes into greater depth about branch & cut as well as other techniques implemented in the ILOG CPLEX MIP Optimizer.

While we have found that the default MIP parameters settings work well for most problems, runtimes can sometime be improved by modifying these settings. This section proposes alternate parameter settings that can help when you are solving difficult MIPs.

Probing

While most of the suggestions in this section are oriented toward overcoming specific obstacles, the *probing* parameter can help in many different ways on difficult models. Probing is a technique that looks at the logical implications of fixing each binary variable to 0 or 1. Probing can be expensive, so this parameter should be used selectively. On models that are in some sense easy, the extra time spent probing may not reduce the overall time enough to be worthwhile. On difficult models, probing may incur very large runtime costs at the beginning and yet pay off with shorter overall runtime. When you are tuning performance, it is usually because the model is difficult, and then probing is worth trying.

When the probing parameter is set to `1` (one), CPLEX performs a limited amount of probing (to limit probing runtime); when set to `2`, the full amount of probing implemented in CPLEX is performed.

To activate probing::

- In the Interactive Optimizer, use the command `set mip strategy probe i`.
- In the Concert Technology Library, set the integer parameter `Probe.`
- In the Callable Library, set the integer parameter `CPX_PARAM_PROBE.`

Too Much Time at Node 0

For some problems, ILOG CPLEX will spend a significant amount of time performing computation at node 0, apart from solving the LP relaxation. While this investment of time normally saves in the overall branch & cut, it does not always do so. Time spent at node 0 can be reduced by two parameters.

First, you can turn off the node heuristic:

- In the Interactive Optimizer, use the command `set mip strategy heuristicfreq -1`.
- In the Concert Technology Library, set the integer parameter `HeurFreq`.
- In the Callable Library, set the integer parameter `CPX_PARAM_HEURFREQ.`

Second, you can choose a less expensive variable selection strategy:

- In the Interactive Optimizer, use the command `set mip strategy variableselect 1` or `4`.
- In the Concert Technology Library, set the integer parameter `VarSel`.
- In the Callable Library, set the integer parameter `CPX_PARAM_VARSEL`.

Time at node 0 can also be consumed by the effort to solve the LP relaxation. Experiment by solving the relaxed problem using each of the LP optimizers. These experiments may suggest a better setting for the `startalgorithm` parameter.

Trouble Finding More than One Feasible Solution

For some models, ILOG CPLEX finds an integer feasible solution early in the process and then does not find a better one for quite a while. One possibility, of course, is that the first feasible solution is optimal. In that case, there are no better solutions.

The more common reason for this behavior, though, is the default best-bound variable selection strategy. This strategy concentrates on exploring nodes that are high in the branch & cut tree for the purpose of proving optimality more quickly.

One easy setting to try is the MIP emphasis parameter. It's described in *Feasibility and Optimality* on page 158. A setting of 1 leads to a greater emphasis on finding feasible solutions during the course of optimization.

If you want to keep the default emphasis on proving optimality, the most useful parameter for altering the default strategy, in the hope of finding new feasible solutions more frequently, is the backtrack parameter. To set its value:

- In the Interactive Optimizer, `set mip strategy backtrack` *`n`*.
- In the Concert Technology Library, set the numeric parameter `BtTol`.

- In the Callable Library, set the double parameter `CPX_PARAM_BTTOL`.).

By setting this value closer to 1.0, you force branch & cut to dive deeper into the tree, where integer feasible solutions are more likely to be found.

Another approach to finding more feasible solutions is to increase the frequency of the node heuristic. To set its value:

- In the Interactive Optimizer, `set mip strategy heuristicfreq` *`i`*.
- In the Concert Technology Library, set the integer parameter `HeurFreq`.
- In the Callable Library, set the integer parameter `CPX_PARAM_HEURFREQ`.

This heuristic can be expensive, so exercise caution when setting this parameter to values less than 10.

A final approach to finding more feasible solutions is to try an alternate node selection strategy. To set the strategy:

- In the Interactive Optimizer, `set mip strategy nodeselect` *`i`*.
- In the Concert Technology Library, set the integer parameter `NodeSel`.
- In the Callable Library, set the integer parameter `CPX_PARAM_NODESEL`.

Values 2 and 3 use node estimates to select nodes and thus sometimes produce more frequent feasible solutions.

Large Number of Unhelpful Cuts

While the cuts added by ILOG CPLEX reduce runtime for most problems, on occasion they can have the opposite effect. If you notice, for example, that ILOG CPLEX adds a large number of cuts at the root, but the objective value does not change significantly, then you may want to experiment with turning off cuts.

- In the Interactive Optimizer, you can turn cuts off selectively (`set mip cuts covers -1`) or all at once (`set mip cuts all -1`).
- In the Component Libraries, set the parameters that control classes of cuts (Table 5.4 on page 160).

Lack of Movement in the Best Node

For some models, the `Best Node` value in the node log changes very slowly or not at all. Runtimes for such models can sometimes be reduced by the variable selection strategy known as *strong branching*. Strong branching explores a set of candidate branching-variables in-depth, performing a limited number of simplex iterations to estimate the effect of branching up or down on each.

Important: *Strong branching consumes significantly more computation time per node than the default variable selection strategy.*

To activate strong branching :

- In the Interactive Optimizer, use the command `set mip strategy variableselect 3`.
- In the Concert Technology Library, set the integer parameter `VarSel`.
- In the Callable Library, set the integer parameter `CPX_PARAM_VARSEL`.

On rare occasions, it can be helpful to modify strong branching limits. If you modify the limit on the size of the candidate list, then strong branching will explore a larger (or smaller) set of candidates. If you modify the limit on strong branching iteration, then strong branching will perform more (or fewer) simplex iterations per candidate. Table 5.9 summarizes those limits and shows the parameter names.

Table 5.9 *Parameters for Limiting Strong Branching*

Limit	Interactive Command	Concert Technology Library Parameter	Callable Library Parameter
size of candidate list	`set mip limits strongcand`	`IloCplex::StrongCandLim`	`CPX_PARAM_STRONGCANDLIM`
iterations per candidate	`set mip limits strongit`	`IloCplex::StrongItLim`	`CPX_PARAM_STRONGITLIM`

Time Wasted on Overly Tight Optimality Criteria

Sometimes ILOG CPLEX finds a good integer solution early, but many additional nodes must be examined to prove that solution is optimal. You can speed up the process in such a case if you are willing to change the optimality tolerance. ILOG CPLEX supports two kinds of tolerance:

- *Relative optimality tolerance* guarantees that a solution lies within a certain percentage of the optimal solution.
- *Absolute optimality tolerance* guarantees that a solution lies within a certain absolute range of the optimal solution.

The default relative optimality tolerance is 0.0001. At this tolerance, the final integer solution is guaranteed to be within 0.01% of the optimal value. Of course, many formulations of integer or mixed integer programs do not require such tight tolerance, so requiring ILOG CPLEX to seek integer solutions that meet this tolerance in those cases is wasted computation. If you can accept greater optimality tolerance in your model, then you should change the parameter to control *relative gap*.

For example, to set the relative gap to one percent:

- In the Interactive Optimizer, use this command: `set mip tolerance mipgap 0.01`.
- In the Concert Technology Library, use the method `IloCplex::setParam(EpGap, 0.01)`.
- In the Callable Library, use the routine `CPXsetdblparam(env, CPX_PARAM_EPGAP, 0.01)`.

If, however, you know that the objective values of your problem are near zero, then you should change the *absolute gap* because percentages of very small numbers are less useful as optimality tolerance.

For example, to change the absolute gap:

- In the Interactive Optimizer, use this command : `set mip tolerance absmipgap 3.0`.
- In the Concert Technology Library, use the method `IloCplex::setParam(EpAGap, 3.0)`.
- In the Callable Library, use the routine `CPXsetdblparam(env, CPX_PARAM_EPAGAP, 3.0)`.

Table 5.10 summarizes the default value and range of absolute and relative gap parameters.

Table 5.10 *Relative, Absolute Gap Parameters (Relative, Absolute Optimality Tolerance)*

	Relative Gap	**Absolute Gap**
Default value	`1e-04`	`1e-6`
Range	`0.0-1.0`	Any positive value
Concert Technology Library parameter	`IloCplex::EpGap`	`IloCplex::EpAGap`
Callable Library parameter	`CPX_PARAM_EPGAP`	`CPX_PARAM_EPAGAP`
Interactive Optimizer option	`mipgap`	`absmipgap`

To speed up the proof of optimality, you can set objective difference parameters, both relative and absolute. Setting these parameters helps when there are many integer solutions with similar objective values. For example, in the Interactive Optimizer, this command `set mip tolerances objdifference 100.0` makes ILOG CPLEX skip any potential solution with its objective value within 100.0 units of the best integer solution so far.

Naturally, since this objective difference setting may make ILOG CPLEX skip an interval where the true integer optimum may be found, the objective difference setting weakens the guarantee of optimality. Table 5.11 summarizes the default value and range of relative and absolute objective difference parameters.

Table 5.11 *Relative and Absolute Objective Difference Parameters*

	Relative Objective Difference	**Absolute Objective Difference**
Default value	`0.0`	`0.0`
Range	`0.0-1.0`	Any value
Concert Technology Library parameter	`IloCplex::RelObjDif`	`IloCplex::ObjDif`
Callable Library parameter	`CPX_PARAM_RELOBJDIF`	`CPX_PARAM_OBJDIF`
Interactive Optimizer option	`relobjdifference`	`objdifference`

Cutoff parameters can also be helpful in restricting the search for optimality. If you know that there are solutions within a certain distance of the initial relaxation of your problem, then you can readily set the upper cutoff parameter for minimization problems and the lower cutoff parameter for maximization problems. For example:

◆ In the Interactive Optimizer, use this command
`set mip tolerances uppercutoff 5000` in a minimization problem, and this one `set mip tolerance lowercutoff 200` in a maximization problem.

◆ When using the Component Libraries, set the parameters `IloCplex::CutUp` or `IloCplex::CutLo` or `CPX_PARAM_CUTUP or CPX_PARAM_CUTLO` and appropriate values.

Table 5.12 summarizes the default value and range of the lower and upper cutoff parameters.

Table 5.12 *Cutoff Parameters*

	Lower cutoff	**Upper cutoff**
Default value	`-1e+75`	`1e+75`
Range	Any value	Any value
Concert Technology Library parameter	`IloCplex::CutLo`	`IloCplex::CutUp`

Table 5.12 Cutoff Parameters (Continued)

	Lower cutoff	Upper cutoff
Callable Library parameter	`CPX_PARAM_CUTLO`	`CPX_PARAM_CUTUP`
Interactive Optimizer option	`lowercutoff`	`uppercutoff`

When you set a MIP cutoff value, ILOG CPLEX searches with the same solution strategy as though it had already found an integer solution, using a node selection strategy that differs from the one it uses before a first solution has been found.

Running Out of Memory

The most common difficulty with MIPs is running out of memory. This problem occurs when the branch & cut tree becomes so large that insufficient memory remains to solve an LP subproblem. As memory gets tight, you may observe warning messages from ILOG CPLEX as it attempts various operations in spite of limited memory. In such a situation, if ILOG CPLEX does not find a solution shortly, it terminates the process with an error message.

The information about a tree that ILOG CPLEX accumulates in memory can be substantial. In particular, ILOG CPLEX saves a basis for every unexplored node. Furthermore, when ILOG CPLEX uses the best bound or best estimate strategies of node selection, the list of unexplored nodes itself can become very long for large or difficult problems. How large the unexplored node list can be depends on the actual amount of memory available, the size of the problem, and algorithm selected.

A less frequent cause of memory consumption is the generation of cutting planes. Gomory fractional cuts, and, in rare instances, Mixed Integer Rounding cuts, are the ones most likely to be dense and thus use significant memory under default/automatic settings. You can try turning off these cuts, or any of the cuts you see listed as being generated for your model (in the cuts summary at the end of the node log), or simply all cuts, through the use of parameter settings discussed in the section on cuts in this manual; doing this carries the risk that this will make the model harder to solve and only delay the eventual exhaustion of available memory during branching.

Certainly, if you increase the amount of available memory, you extend the problem-solving capability of ILOG CPLEX. Unfortunately, when a problem fails because of insufficient memory, it is difficult to project how much further the process needed to go and how much more memory is needed to solve the problem. For these reasons, the following suggestions aim at avoiding memory failure whenever possible and recovering gracefully otherwise.

Reset the Tree Memory Parameter

To avoid a failure due to running out of memory, we recommend setting the working memory parameter to instruct CPLEX to begin using disk for storage of nodes before it consumes all available memory.

To set the working memory parameter:

- In the Interactive Optimizer, use the command `set workmem n` using a value `n` that is smaller than the total available memory in megabytes.
- For the Component Libraries, set the parameter `IloCplex::WorkMem` or CPX_PARAM_WORKMEM.

Because the storage of nodes can require a lot of space, it may also be advisable to set a tree limit on the size of the entire tree being stored so that not all of your disk will be filled up with working storage. The call to the MIP optimizer will be stopped once the size of the tree exceeds the value of the tree limit parameter. Under default settings the limit is infinity (1e+75), but you can set it to a lower value (in megabytes):

To set the tree limit parameter:

- In the Interactive Optimizer, use the command `set mip limits treememory`.
- For the Component Libraries, set the parameter `IloCplex::TreLim`, or `CPX_PARAM_TRELIM`.

Write a Tree File and Restart

On some platforms, even when the current tree size is within system limits, memory fragmentation may be so great that performance becomes poor. To overcome that kind of fragmentation, we recommend that you stop optimization, write a tree file (using the TRE format), exit ILOG CPLEX, restart it, read in the model and tree file, and continue optimization then.

Use Node Files for Storage

ILOG CPLEX offers a node file storage feature to store some parts of the branch & cut tree in files. If you use this feature, CPLEX will be able to explore more nodes within a smaller amount of computer memory.This feature includes several options to reduce the use of physical memory, and it entails a very small increase in runtime, so it has less overall impact on system resources. Node file storage offers a much better option than relying on swap space.

This feature is especially helpful when you are using steepest-edge pricing as the subproblem simplex pricing strategy because pricing information itself consumes a great deal of memory.

There are several parameters that control the use of node files. They are summarized as follows, and described in detail in the next paragraphs:

Table 5.13 *Node File Control Parameters*

Interactive Optimizer	**Concert Technology Library**	**Callable Library**
mip limits treememory	`IloCplex::TreLim`	`CPX_PARAM_TRELIM`
mip strategy file	`IloCplex::NodeFileInd`	`CPX_PARAM_NODEFILEIND`
workdir	`IloCplex::WorkDir`	`CPX_PARAM_WORKDIR`
workmem	`IloCplex::WorkMem`	`CPX_PARAM_WORKMEM`

ILOG CPLEX invokes node file storage when it reaches the working memory limit. By default, the limit is 128 (megabytes).

To set a limit on the size of the branch & cut tree held in memory:

- ◆ In the Interactive Optimizer, use the command `set workmem` *`n`*, substituting a value for *`n`*.
- ◆ When using the Component Libraries, set the parameter `IloCplex::WorkMem` or `CPX_PARAM_WORKMEM`.

ILOG CPLEX uses node file storage most effectively when the amount of working memory is reasonably large so that it does not have to create node files too frequently. A reasonable amount is to use approximately half the memory, but no more than 32 megabytes. Higher values result in only marginally improved efficiency.

When tree storage size exceeds the limit defined by `IloCplex::WorkMem` / `CPX_PARAM_WORKMEM`, what happens next is determined by the setting of `IloCplex::NodeFileInd` / `CPX_PARAM_NODEFILEIND`. If the latter parameter is set to zero, then optimization proceeds with the tree stored in memory until CPLEX reaches the tree memory limit (`IloCplex::TreLim / CPX_PARAM_TRELIM`). If the parameter is set to 1 (the default), then a very fast compression algorithm is used on the nodes to try to conserve memory, without resorting to writing the node files to disk. If the parameter is set to 2, then node files are written to disk. If the parameter is set to 3, then nodes are both compressed (as in option 1) and written to disk (as in option 2). Thus, regardless of the setting of `IloCplex::NodeFileInd /CPX_PARAM_NODEFILEIND`, CPLEX will stop the optimization when the total memory used to store the tree exceeds the tree memory limit.

Table 5.14 *Values for the Node File Storage Parameter*

Value	**Meaning**	**Comments**
`0`	no node files	optimization continues
`1`	node file in memory and compressed	optimization continues (default)

Table 5.14 *Values for the Node File Storage Parameter (Continued)*

Value	Meaning	Comments
2	node file on disk	files created in temporary directory
3	node file on disk and compressed	files created in temporary directory

In cases where node files are written to disk, CPLEX will create a temporary subdirectory under the directory specified by the `IloCplex::WorkDir` / `CPX_PARAM_WORKDIR` parameter. The directory named by this parameter must exist before CPLEX attempts to create node files. By default, the value of this parameter is ".", which means the current working directory.

ILOG CPLEX creates the temporary directory by means of system calls. If the system environment variable is set (on Windows 95 or NT, the environment variable `TMP`; on UNIX platforms, the environment variable `TMPDIR`), then the system ignores the ILOG CPLEX node-file directory parameter and creates the temporary node-file directory in the location indicated by its system environment variable. Furthermore, if the directory specified in the ILOG CPLEX node-file directory parameter is invalid (for example, if it contains illegal characters, or if the directory does not allow write access), then the system chooses a location according to its own logic.

The temporary directory created for node file storage will have a name prefixed by `cpx`. The files within it will also have names prefixed by `cpx`.

ILOG CPLEX automatically removes the files and their temporary directory when it frees the branch & cut tree:

- in the Interactive Optimizer,
  - at problem modification;
  - at normal termination;
- from the Concert Technology Library,
  - when you call `env.end()`
- from the Callable Library,
  - when you call a problem modification routine;
  - when you call `CPXfreeprob()`.

If a program terminates abnormally, the files are not removed.

Node files may grow very large. You can limit their size by setting the file limit parameter:

- In the Interactive Optimizer, use the command `set workmem` with any positive value.
- In the Concert Technology Library, use the method `IloCplex::setParam(Workmem, n)`, and in the Callable Library, use the routine `CPXsetdblparam(env, CPX_PARAM_WORKMEM, n)` where *n* is any positive value. The default value is *128 (megabytes).*

When ILOG CPLEX uses node-file storage, the sequence of nodes processed may differ from the sequence in which nodes are processed without node-file storage. Nodes in node-file storage are not accessible to user-written callback routines.

Change Algorithms

The best approach to reduce memory use is to modify the solution process. Here are some ways to do so:

- Switch to a higher backtracking parameter, as suggested on page 176.
- Switch the node selection strategy to best estimate, or more drastically to depth-first, as explained on page 177. Depth-first search rarely generates a long, memory-consuming list of unexplored nodes since ILOG CPLEX dives deeply into the tree instead of jumping around. A narrowly focused search, like depth-first, also often results in faster processing times for individual nodes. However, overall efficiency is sometimes worse than with best-bound node selection because each branch is searched exhaustively to its deepest level before it is fathomed in favor of better branches.
- Another memory-conserving strategy is to use strong branching for variable selection. Strong branching requires substantial computational effort at each node to determine the best branching variable. As a result, it generates fewer nodes and thus makes less overall demand on memory. Often, strong branching is faster as well.
- On some problems, the automatic generation of cuts results in excessive memory use with little benefit in speed. In such cases, we recommend that you turn off cut generation.
  - In the Interactive Optimizer, use the commands `set mip cuts all -1` to turn off all cuts. Use `set mip cuts` *`class`* `-1` (where *`class`* may be `cliques`, `covers` etc.) to turn off individual classes of cuts.
  - In the Component Libraries, cuts may be turned off only by class; use the method `IloCplex::setParam()` or the routine `CPXsetintparam()` with the appropriate parameter to indicate which class of cuts to turn off (`Cliques` / `CPX_PARAM_CLIQUES`, `Covers` / `CPX_PARAM_COVERS`, `etc`) and the value `-1` each time.

  See Table 5.4 on page 160 for a complete list of available cuts.

Difficulty Solving Subproblems

There are classes of MIPs that produce very difficult subproblems, for example, if the subproblems are dual degenerate. In such a case, an alternative optimizer, such as the primal simplex or the primal-dual barrier optimizer, may be better suited to your problem than the default dual simplex optimizer for subproblems.

Overcoming Degeneracy

If the subproblems are dual degenerate, then consider using the primal simplex optimizer for the subproblems. Set the subalgorithm parameter, as explained in *Subalgorithm Parameter* on page 188, to use the primal simplex optimizer.

Another effective strategy in overcoming dual degeneracy is to permanently perturb the problem. For subproblems that are dual degenerate, in the Interactive Optimizer, write out the perturbed problem as a DPE file with the command `write` *`filename`*`.dpe` substituting an appropriate file name. (A `.dpe` file is saved as a binary SAV format file.) Then you can read the saved file back in and solve it. The subproblem should then solve more cleanly and quickly.

In the case of DPE files solved by the dual simplex optimizer, any integer solution is also guaranteed to be an integer-feasible solution to the original problem. In most cases, the solution will be optimal or near-optimal as well.

Shortening Long Solution Times

If subproblems are taking many iterations per node to solve, consider using a stronger dual pricing algorithm, such as dual steepest-edge pricing.

In case you have selected the primal-dual barrier optimizer to solve the initial LP relaxation, you may want to apply it to the subproblems in one of two ways:

- barrier with crossover
  - in the Interactive Optimizer use, `set mip strategy subalgorithm 4`
  - in the Concert Technology Library, use the method `IloCplex::setNodeAlgorithm(Barrier)`
  - or `CPXsetintparam(env, CPX_PARAM_SUBALG, CPX_NODEALG_BARRIER)`.

  This choice applies the primal-dual barrier optimizer to all subproblems.
- dual to limit, then barrier
  - in the Interactive Optimizer use, `set mip strategy subalgorithm 5`
  - in the Concert Technology Library, use the method `IloCplex::setNodeAlgorithm(DualBarrier)`
  - or `CPXsetintparam(env, CPX_PARAM_SUBALG, CPX_NODEALG_DUAL)`

Recognizing that the barrier optimizer does not utilize a basis, this choice lets the dual simplex optimizer run for a predetermined number of iterations and then switches to the barrier optimizer for the subproblem. To use this choice, you need to set the simplex iteration limit to a reasonably low number of dual iterations.

If you limit the number of simplex iterations, the limit applies to all invocations of simplex optimizers, except crossover. Since the dual simplex optimizer will most often be the best method, try to specify a sufficient number of iterations before you force the switch to the barrier optimizer.

Subproblem Optimization

In some problems, you can improve performance by evaluating how the LP subproblems are solved at the nodes in the branch & cut tree, and then possibly modifying the choice of algorithm to solve them. As we mentioned in *Preprocessing: Presolver and Aggregator* on page 163, you can control which algorithm ILOG CPLEX applies to the initial relaxation of your problem separately from your control of which algorithm ILOG CPLEX applies to other subproblems. Table 5.15 summarizes the commands to control those two parameters. The following sections explain those parameters more fully.

Table 5.15 *Parameters for MIP Initial Relaxation and Subproblems*

Interactive command	Callable Library parameter	Applies to
`set mip strategy startalgorithm`	`CPX_PARAM_STARTALG`	initial relaxation
`set mip strategy subalgorithm`	`CPX_PARAM_SUBALG`	subproblems

Start-Algorithm Parameter

The start-algorithm parameter indicates the algorithm for ILOG CPLEX to use on the *initial* subproblem. In a typical MIP, that initial subproblem is usually the linear relaxation of the original MIP. By default, ILOG CPLEX starts the initial subproblem with the dual simplex optimizer. You may have information about your problem that indicates another optimizer could be more efficient. Table 5.16 summarizes the values available for the start-algorithm parameter.

To set this parameter:

- In the Interactive Optimizer, use the command `set mip strategy startalgorithm` with the value to indicate the optimizer you want.
- In the Concert Technology library, use the method `IloCplex::setRootAlgorithm()` and the appropriate algorithm enumeration value.
- In the Callable Library, use the routine `CPXsetintparam()` with the parameter `CPX_PARAM_STARTALG`, and the appropriate symbolic constant.

Table 5.16 *Values of Start-Algorithm and Sub-Algorithm Parameters*

Concert Technology Library Enumeration	Callable Library Symbolic Constant	Value	Calls this Optimizer
`IloCplex::Primal`	`CPX_NODEALG_PRIMAL`	1	primal simplex
`IloCplex::Dual`	`CPX_NODEALG_DUAL`	2	dual simplex (default)
`IloCplex::Network`	`CPX_NODEALG_HYBNETOPT`	3	network simplex
`IloCplex::Barrier`	`CPX_NODEALG_HYBBAROPT`	4	barrier with crossover (if licensed)
`IloCplex::DualBarrier`	`CPX_NODEALG_DUAL_HYBBAROPT`	5	dual simplex to iteration limits, then barrier (if licensed)
	`CPX_NODEALG_BARRIER`	6	barrier without crossover (if licensed)

Crossover Parameter

To control the kind of crossover used by the barrier optimizer for MIP subproblems, in the Interactive Optimizer, use the command `set mip strategy crossover` *`i`* substituting a value to indicate which optimizer to call at crossover. From the callable Library, use the routine `CPXsetintparam()` with the parameter `CPX_PARAM_MIPHYBALG` and a crossover value. Table 5.17 lists the acceptable values for this crossover parameter.

Table 5.17 *Crossover parameter values used for MIP subproblems*

Value	Calls this Optimizer
1 (default)	primal crossover
2	dual crossover

Subalgorithm Parameter

The subalgorithm parameter indicates the algorithm for ILOG CPLEX to use on *subsequent* subproblems. By default, ILOG CPLEX applies the dual simplex optimizer to subproblems, but again, you may have information about your problem that tells you another optimizer could be more efficient. To specify a subalgorithm in the Interactive Optimizer, use the command `set mip strategy subalgorithm` with the value to indicate the optimizer you want. In the Concert Technology library use the method `IloCplex::setNodeAlgorithm()` and the appropriate algorithm enumeration value. In the Callable Library, use the routine `CPXsetintparam()` with the parameter `CPX_PARAM_SUBALG`, and the appropriate symbolic constant. The values and symbolic constants are the same for the subalgorithm parameter as for the start-algorithm parameter in Table 5.16 on page 188.

Example: Optimizing a Basic MIP Problem

This example illustrates how to optimize a MIP with the ILOG CPLEX Component Libraries.

Complete Program: ilomipex1.cpp

The example derives from `ilolpex8.cpp`. Here are the differences between that linear program and this mixed integer program:

- ◆ The problem to solve is slightly different. It appears in *Sample: Stating a MIP Problem* on page 152.
- ◆ The routine`populatebyrow()` added the variables, objective, and constraints to the model created by the method `IloModel model(env)`.

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

ILOSTLBEGIN

static void
   populatebyrow(IloModel model, IloNumVarArray var, IloRangeArray con);

int
main (void) {
   IloEnv env;
   try {
      IloModel model(env);

      IloNumVarArray var(env);
      IloRangeArray con(env);
      populatebyrow (model, var, con);

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

      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      env.out() << "Values        = " << vals << endl;
      cplex.getSlacks(vals, con);
      env.out() << "Slacks        = " << vals << endl;

      cplex.exportModel("mipex1.lp");
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();
   return 0;

}  // END main


static void
populatebyrow (IloModel model, IloNumVarArray x, IloRangeArray c)
{
   IloEnv env = model.getEnv();

   x.add(IloNumVar(env, 0.0, 40.0));
   x.add(IloNumVar(env));
   x.add(IloNumVar(env));
   x.add(IloNumVar(env, 2.0, 3.0, ILOINT));
   model.add(IloMaximize(env, x[0] + 2 * x[1] + 3 * x[2] + x[3]));

   c.add( - x[0] +     x[1] + x[2] + 10 * x[3] <= 20);
   c.add(   x[0] - 3 * x[1] + x[2]             <= 30);
   c.add(              x[1]        - 3.5* x[3] == 0);
   model.add(c);

}  // END populatebyrow
```

Complete Program: mipex1.c

The example derives from `lpex8.c`. Here are the differences between that linear program and this mixed integer program:

- The problem to solve is slightly different. It appears in *Sample: Stating a MIP Problem* on page 152.
- The routine `setproblemdata()` has a parameter, `ctype`, to set the types of the variables to indicate which ones must assume integer values. The routine `CPXcopyctype()` associates this data with the problem that `CPXcreateprob()` creates.
- The example calls `CPXmipopt()` to optimize the problem, not `CPXprimopt()`, of course. `CPXmipopt()` solves MIPs.
- The example calls the routines `CPXgetstat()`, `CPXgetmipobjval()`, `CPXgetmipx()`, and `CPXgetmipslack()` (instead of `CPXsolution()`) to get a solution.

We do not get dual variables this way. If we want dual variables, we must do the following:

- Use `CPXchgprobtype()` to change the problem type to `CPXPROB_FIXED`.
- Then call `CPXprimopt()` to optimize that problem.

Then use `CPXsolution()` to get a solution to the *fixed* problem.

```
#include <ilcplex/cplex.h>
#include <stdlib.h>

/* Bring in the declarations for the string functions */

#include <string.h>

/* Include declaration for function at end of program */

#ifndef  CPX_PROTOTYPE_MIN

static int
   setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                   int *objsen_p, double **obj_p, double **rhs_p,
                   char **sense_p, int **matbeg_p, int **matcnt_p,
                   int **matind_p, double **matval_p,
                   double **lb_p, double **ub_p, char **ctype_p);

static void
   free_and_null (char **ptr);

#else

static int
   setproblemdata ();

static void
   free_and_null ();

#endif


/* The problem we are optimizing will have 2 rows, 3 columns
   and 6 nonzeros.  */

#define NUMROWS    3
#define NUMCOLS    4
#define NUMNZ      9

#ifndef  CPX_PROTOTYPE_MIN
int
main (void)
#else
int
```

```
main ()
#endif
{
/* Declare pointers for the variables and arrays that will contain
   the data which define the LP problem.  The setproblemdata() routine
   allocates space for the problem data.  */

   char     *probname = NULL;
   int      numcols;
   int      numrows;
   int      objsen;
   double   *obj = NULL;
   double   *rhs = NULL;
   char     *sense = NULL;
   int      *matbeg = NULL;
   int      *matcnt = NULL;
   int      *matind = NULL;
   double   *matval = NULL;
   double   *lb = NULL;
   double   *ub = NULL;
   char     *ctype = NULL;

   /* Declare and allocate space for the variables and arrays where we will
      store the optimization results including the status, objective value,
      variable values, and row slacks. */

   int      solstat;
   double   objval;
   double   x[NUMCOLS];
   double   slack[NUMROWS];


   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status;
   int           i, j;
   int           cur_numrows, cur_numcols;

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no output,
      so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
      be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

   if ( env == NULL ) {
      char  errmsg[1024];
      fprintf (stderr, "Could not open CPLEX environment.\n");
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, "%s", errmsg);
      goto TERMINATE;
   }

   /* Turn on output to the screen */

   status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
   if ( status ) {
      fprintf (stderr,
               "Failure to turn on screen indicator, error %d.\n", status);
      goto TERMINATE;
   }

   /* Fill in the data for the problem.  */

   status = setproblemdata (&probname, &numcols, &numrows, &objsen, &obj,
                            &rhs, &sense, &matbeg, &matcnt, &matind, &matval,
                            &lb, &ub, &ctype);
   if ( status ) {
      fprintf (stderr, "Failed to build problem data arrays.\n");
      goto TERMINATE;
   }

   /* Create the problem. */

   lp = CPXcreateprob (env, &status, probname);

   /* A returned pointer of NULL may mean that not enough memory
      was available or there was some other problem.  In the case of
      failure, an error message will have been written to the error
      channel from inside CPLEX.  In this example, the setting of
      the parameter CPX_PARAM_SCRIND causes the error message to
      appear on stdout.  */

   if ( lp == NULL ) {
      fprintf (stderr, "Failed to create LP.\n");
      goto TERMINATE;
   }

   /* Now copy the problem data into the lp */

   status = CPXcopylp (env, lp, numcols, numrows, objsen, obj, rhs,
                       sense, matbeg, matcnt, matind, matval,
                       lb, ub, NULL);

   if ( status ) {
      fprintf (stderr, "Failed to copy problem data.\n");
      goto TERMINATE;
   }

   /* Now copy the ctype array */
```

```
status = CPXcopyctype (env, lp, ctype);
if ( status ) {
   fprintf (stderr, "Failed to copy ctype\n");
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */

status = CPXmipopt (env, lp);
if ( status ) {
   fprintf (stderr, "Failed to optimize MIP.\n");
   goto TERMINATE;
}

solstat = CPXgetstat (env, lp);

/* Write the output to the screen. */

printf ("\nSolution status = %d\n", solstat);

status = CPXgetmipobjval (env, lp, &objval);
if ( status ) {
   fprintf (stderr,"No MIP objective value available.  Exiting...\n");
   goto TERMINATE;
}

printf ("Solution value  = %f\n\n", objval);

/* The size of the problem should be obtained by asking CPLEX what
   the actual size is, rather than using what was passed to CPXcopylp.
   cur_numrows and cur_numcols store the current number of rows and
   columns, respectively.  */

cur_numrows = CPXgetnumrows (env, lp);
cur_numcols = CPXgetnumcols (env, lp);

status = CPXgetmipx (env, lp, x, 0, cur_numcols-1);
if ( status ) {
   fprintf (stderr, "Failed to get optimal integer x.\n");
   goto TERMINATE;
}

status = CPXgetmipslack (env, lp, slack, 0, cur_numrows-1);
if ( status ) {
   fprintf (stderr, "Failed to get optimal slack values.\n");
   goto TERMINATE;
}

for (i = 0; i < cur_numrows; i++) {
   printf ("Row %d:  Slack = %10f\n", i, slack[i]);
}

for (j = 0; j < cur_numcols; j++) {
      printf ("Column %d:  Value = %10f\n", j, x[j]);
   }

   /* Finally, write a copy of the problem to a file. */

   status = CPXwriteprob (env, lp, "mipex1.lp", NULL);
   if ( status ) {
      fprintf (stderr, "Failed to write LP to disk.\n");
      goto TERMINATE;
   }

TERMINATE:

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   /* Free up the problem data arrays, if necessary. */

   free_and_null ((char **) &probname);
   free_and_null ((char **) &obj);
   free_and_null ((char **) &rhs);
   free_and_null ((char **) &sense);
   free_and_null ((char **) &matbeg);
   free_and_null ((char **) &matcnt);
   free_and_null ((char **) &matind);
   free_and_null ((char **) &matval);
   free_and_null ((char **) &lb);
   free_and_null ((char **) &ub);
   free_and_null ((char **) &ctype);

   return (status);

}  /* END main */


/* This function fills in the data structures for the mixed integer program:

      Maximize
       obj: x1 + 2 x2 + 3 x3 + x4
      Subject To
       c1: - x1 + x2 + x3 + 10x4  <= 20
       c2: x1 - 3 x2 + x3         <= 30
       c3:       x2       - 3.5x4  = 0
      Bounds
       0 <= x1 <= 40
       2 <= x4 <= 3
      Integers
        x4
      End
 */


#ifndef  CPX_PROTOTYPE_MIN
static int
setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                int *objsen_p, double **obj_p, double **rhs_p,
                char **sense_p, int **matbeg_p, int **matcnt_p,
                int **matind_p, double **matval_p,
                double **lb_p, double **ub_p, char **ctype_p)
#else
static int
setproblemdata (probname_p, numcols_p, numrows_p, objsen_p, obj_p,
                rhs_p, sense_p, matbeg_p, matcnt_p, matind_p, matval_p,
                lb_p, ub_p, ctype_p)
char    **probname_p;
int     *numcols_p;
int     *numrows_p;
int     *objsen_p;
double  **obj_p;
double  **rhs_p;
char    **sense_p;
int     **matbeg_p;
int     **matcnt_p;
int     **matind_p;
double  **matval_p;
double  **lb_p;
double  **ub_p;
char    **ctype_p;
#endif
{
   char     *zprobname = NULL;     /* Problem name <= 16 characters */
```

```
double   *zobj = NULL;
double   *zrhs = NULL;
char     *zsense = NULL;
int      *zmatbeg = NULL;
int      *zmatcnt = NULL;
int      *zmatind = NULL;
double   *zmatval = NULL;
double   *zlb = NULL;
double   *zub = NULL;
char     *zctype = NULL;
int      status = 0;

zprobname = (char *) malloc (16 * sizeof(char));
zobj      = (double *) malloc (NUMCOLS * sizeof(double));
zrhs      = (double *) malloc (NUMROWS * sizeof(double));
zsense    = (char *) malloc (NUMROWS * sizeof(char));
zmatbeg   = (int *) malloc (NUMCOLS * sizeof(int));
zmatcnt   = (int *) malloc (NUMCOLS * sizeof(int));
zmatind   = (int *) malloc (NUMNZ * sizeof(int));
zmatval   = (double *) malloc (NUMNZ * sizeof(double));
zlb       = (double *) malloc (NUMCOLS * sizeof(double));
zub       = (double *) malloc (NUMCOLS * sizeof(double));
zctype    = (char *) malloc (NUMCOLS * sizeof(char));

if ( zprobname == NULL || zobj    == NULL ||
     zrhs      == NULL || zsense  == NULL ||
     zmatbeg   == NULL || zmatcnt == NULL ||
     zmatind   == NULL || zmatval == NULL ||
     zlb       == NULL || zub     == NULL ||
     zctype    == NULL                       )  {
   status = 1;
   goto TERMINATE;
}

strcpy (zprobname, "example");

/* The code is formatted to make a visual correspondence
   between the mathematical linear program and the specific data
   items.   */

  zobj[0]  = 1.0;   zobj[1]   = 2.0;  zobj[2]    = 3.0;    zobj[3] = 1.0;

zmatbeg[0] = 0;    zmatbeg[1] = 2;    zmatbeg[2] = 5;   zmatbeg[3] = 7;
zmatcnt[0] = 2;    zmatcnt[1] = 3;    zmatcnt[2] = 2;   zmatcnt[3] = 2;

zmatind[0] = 0;    zmatind[2] = 0;    zmatind[5] = 0;   zmatind[7] = 0;
zmatval[0] = -1.0; zmatval[2] = 1.0;  zmatval[5] = 1.0; zmatval[7] = 10.0;

zmatind[1] = 1;    zmatind[3] = 1;    zmatind[6] = 1;
zmatval[1] = 1.0;  zmatval[3] = -3.0; zmatval[6] = 1.0;

                   zmatind[4] = 2;                      zmatind[8] = 2;
                   zmatval[4] = 1.0;                   zmatval[8] = -3.5;

   zlb[0] = 0.0;   zlb[1] = 0.0;      zlb[2] = 0.0;     zlb[3] = 2.0;
   zub[0] = 40.0;  zub[1] = CPX_INFBOUND; zub[2] = CPX_INFBOUND; zub[3] = 3.0;

    zctype[0] = ’C’;    zctype[1] = ’C’;   zctype[2] = ’C’;   zctype[3] = ’I’;

  /* The right-hand-side values don’t fit nicely on a line above.  So put
     them here.  */

   zsense[0] = ’L’;
   zrhs[0]   = 20.0;

   zsense[1] = ’L’;
   zrhs[1]   = 30.0;

   zsense[2] = ’E’;
   zrhs[2]   = 0.0;

TERMINATE:

   if ( status ) {
      free_and_null ((char **) &zprobname);
      free_and_null ((char **) &zobj);
      free_and_null ((char **) &zrhs);
      free_and_null ((char **) &zsense);
      free_and_null ((char **) &zmatbeg);
      free_and_null ((char **) &zmatcnt);
      free_and_null ((char **) &zmatind);
      free_and_null ((char **) &zmatval);
      free_and_null ((char **) &zlb);
      free_and_null ((char **) &zub);
      free_and_null ((char **) &zctype);
   }
   else {
      *numcols_p   = NUMCOLS;
      *numrows_p   = NUMROWS;
      *objsen_p    = CPX_MAX;   /* The problem is maximization */

      *probname_p  = zprobname;
      *obj_p       = zobj;
      *rhs_p       = zrhs;
      *sense_p     = zsense;
      *matbeg_p    = zmatbeg;
      *matcnt_p    = zmatcnt;
      *matind_p    = zmatind;
      *matval_p    = zmatval;
      *lb_p        = zlb;
      *ub_p        = zub;
      *ctype_p     = zctype;
   }
   return (status);
```

```
}  /* END setproblemdata */



/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

#ifndef  CPX_PROTOTYPE_MIN
static void
free_and_null (char **ptr)
#else
static void
free_and_null (ptr)
char  **ptr;
#endif
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */
```

Example: Reading a MIP Problem from a File

This example shows you how to solve a MIP with the Component Libraries when the problem data is stored in a file.

Example: ilomipex2.cpp

This example derives from `ilolpex2.cpp`, an LP explained in the manual *Getting Started with ILOG CPLEX*. That LP example differs from this MIP example in these ways:

◆ This example solves only MIPs, so it calls only `IloCplex::solve()`, and its command line does not require the user to indicate an optimizer.

◆ This example doesn't generate or print a basis.

Like other applications based on the ILOG CPLEX Concert Technology Library, this one uses `IloEnv env` to initialize the Concert Technology environment and `IloModel model(env)` to create a problem object. Before it ends, it calls `env.end` to free the environment.

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

static void usage (const char *progname);

int
main (int argc, char **argv)
```

```
{
   IloEnv env;
   try {
      IloModel model(env);
      IloCplex cplex(env);

      if ( argc != 2 ) {
         usage (argv[0]);
         throw(-1);
      }

      IloObjective   obj;
      IloNumVarArray var(env);
      IloRangeArray  rng(env);
      cplex.importModel(model, argv[1], obj, var, rng);

      cplex.extract(model);
      cplex.solve();

      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      env.out() << "Values        = " << vals << endl;
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();
   return 0;
}  // END main


static void usage (const char *progname)
{
   cerr << "Usage: " << progname << " filename" << endl;
   cerr << "   where filename is a file with extension " << endl;
   cerr << "      MPS, SAV, or LP (lower case is allowed)" << endl;
   cerr << " Exiting..." << endl;
} // END usage
```

Example: mipex2.c

The example derives from `lpex2.c`, an LP explained in the manual *Getting Started with ILOG CPLEX*. That LP example differs from this MIP example in these ways:

- This example solves only MIPs, so it calls only `CPXmipopt()`, and its command line does not require the user to indicate an optimizer.
- This example calls `CPXgetstat()`, `CPXgetmipobjval()`, and `CPXgetmipx()` to get a solution. It doesn't generate or print a basis.

Like other applications based on the ILOG CPLEX Callable Library, this one calls `CPXopenCPLEX()` to initialize the ILOG CPLEX environment; it sets the screen-indicator parameter to direct output to the screen and calls `CPXcreateprob()` to create a problem object. Before it ends, it calls `CPXfreeprob()` to free the space allocated to the problem object and `CPXcloseCPLEX()` to free the environment.

```
#include <ilcplex/cplex.h>

/* Bring in the declarations for the string and character functions
   and malloc */

#include <ctype.h>
#include <stdlib.h>
#include <string.h>

/* Include declarations for functions in this program */

#ifndef  CPX_PROTOTYPE_MIN

static void
   free_and_null (char **ptr),
   usage         (char *progname);

#else

static void
   free_and_null (),
   usage         ();

#endif

#ifndef  CPX_PROTOTYPE_MIN
int
main (int argc, char *argv[])
#else
int
main (argc, argv)
int   argc;
char  *argv[];
#endif
{
   /* Declare and allocate space for the variables and arrays where we will
      store the optimization results including the status, objective value,
      and variable values. */
```

```
int      solstat;
double   objval;
double   *x     = NULL;

CPXENVptr     env = NULL;
CPXLPptr      lp = NULL;
int           status;
int           j;
int           cur_numcols;

/* Check the command line arguments */

if ( argc != 2 ) {
   usage (argv[0]);
   goto TERMINATE;
}

/* Initialize the CPLEX environment */

env = CPXopenCPLEX (&status);

/* If an error occurs, the status value indicates the reason for
   failure.  A call to CPXgeterrorstring will produce the text of
   the error message.  Note that CPXopenCPLEX produces no output,
   so the only way to see the cause of the error is to use
   CPXgeterrorstring.  For other CPLEX routines, the errors will
   be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

if ( env == NULL ) {
   char  errmsg[1024];
   fprintf (stderr, "Could not open CPLEX environment.\n");
   CPXgeterrorstring (env, status, errmsg);
   fprintf (stderr, "%s", errmsg);
   goto TERMINATE;
}

/* Turn on output to the screen */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
if ( status ) {
   fprintf (stderr,
            "Failure to turn on screen indicator, error %d.\n", status);
   goto TERMINATE;
}

/* Create the problem, using the filename as the problem name */

lp = CPXcreateprob (env, &status, argv[1]);

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.  Note that most CPLEX routines return
   an error code to indicate the reason for failure.   */
```

```
if ( lp == NULL ) {
   fprintf (stderr, "Failed to create LP.\n");
   goto TERMINATE;
}

/* Now read the file, and copy the data into the created lp */

status = CPXreadcopyprob (env, lp, argv[1], NULL);
if ( status ) {
   fprintf (stderr, "Failed to read and copy the problem data.\n");
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */

status = CPXmipopt (env, lp);

if ( status ) {
   fprintf (stderr, "Failed to optimize MIP.\n");
   goto TERMINATE;
}

solstat = CPXgetstat (env, lp);
printf ("Solution status %d.\n", solstat);

status  = CPXgetmipobjval (env, lp, &objval);

if ( status ) {
   fprintf (stderr,"Failed to obtain objective value.\n");
   goto TERMINATE;
}

printf ("Objective value %.10g\n", objval);

/* The size of the problem should be obtained by asking CPLEX what
   the actual size is. cur_numcols stores the current number
   of columns. */

cur_numcols = CPXgetnumcols (env, lp);

/* Allocate space for solution */

x = (double *) malloc (cur_numcols*sizeof(double));

if ( x == NULL ) {
   fprintf (stderr, "No memory for solution values.\n");
   goto TERMINATE;
}

status = CPXgetmipx (env, lp, x, 0, cur_numcols-1);
if ( status ) {
   fprintf (stderr, "Failed to obtain solution.\n");
   goto TERMINATE;
}

/* Write out the solution */
```

```
   for (j = 0; j < cur_numcols; j++) {
      printf ( "Column %d:  Value = %17.10g\n", j, x[j]);
   }


TERMINATE:

   /* Free up the solution */

   free_and_null ((char **) &x);

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
      char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   return (status);

}  /* END main */


/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

#ifndef  CPX_PROTOTYPE_MIN
static void
free_and_null (char **ptr)
#else
static void
free_and_null (ptr)
char  **ptr;
#endif
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
```

```
} /* END free_and_null */


#ifndef  CPX_PROTOTYPE_MIN
static void
usage (char *progname)
#else
static void
usage (progname)
char *progname;
#endif
{
   fprintf (stderr,"Usage: %s filename\n", progname);
   fprintf (stderr,"   where filename is a file with extension \n");
   fprintf (stderr,"      MPS, SAV, or LP (lower case is allowed)\n");
   fprintf (stderr,"  This program uses the CPLEX MIP optimizer.\n");
   fprintf (stderr," Exiting...\n");
} /* END usage */
```

Example: Using SOS and Priority

This example illustrates how to use SOS and priority orders.

Example: ilomipex3.cpp

It derives from `ilomipex1.cpp`. The differences between that simpler MIP example and this one are:

- The problem solved is slightly different so the output is interesting. The actual SOS and priority order that the example implements are arbitrary; they do not necessarily represent good data for this problem.
- The routine `setPriorities()` sets the SOS and priority order:

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

ILOSTLBEGIN

static void
   populatebyrow(IloModel model, IloNumVarArray var, IloRangeArray con);

int
main (void) {
   IloEnv env;
   try {
      IloModel model(env);

      IloNumVarArray var(env);
      IloRangeArray con(env);
      populatebyrow (model, var, con);
      IloCplex cplex(model);
      IloNumVarArray ordvar(env, 2, var[1], var[3]);
      IloNumArray    ordpri(env, 2, 8.0, 7.0);
      cplex.setPriorities (ordvar, ordpri);
      cplex.setDirection(var[1], IloCplex::BranchUp);
      cplex.setDirection(var[3], IloCplex::BranchDown);
      cplex.solve();

      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      env.out() << "Values        = " << vals << endl;
      cplex.getSlacks(vals, con);
      env.out() << "Slacks        = " << vals << endl;

      cplex.exportModel("mipex3.lp");
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();
   return 0;

}  // END main


static void
populatebyrow (IloModel model, IloNumVarArray x, IloRangeArray c)
{
   IloEnv env = model.getEnv();

   x.add(IloNumVar(env, 0.0, 40.0));
   x.add(IloNumVar(env, 0.0, IloInfinity, ILOINT));
   x.add(IloNumVar(env, 0.0, IloInfinity, ILOINT));
   x.add(IloNumVar(env, 2.0, 3.0, ILOINT));
   model.add(IloMaximize(env, x[0] + 2 * x[1] + 3 * x[2] + x[3]));

   c.add( - x[0] +     x[1] + x[2] + 10 * x[3] <= 20);
   c.add(   x[0] - 3 * x[1] + x[2]             <= 30);
   c.add(              x[1]       - 3.5* x[3] == 0);
   model.add(c);

   model.add(IloSOS1(model.getEnv(),
                  IloNumVarArray(model.getEnv(), 2, x[2], x[3]),
                  IloNumArray(env, 2, 25.0, 18.0)    ));
```

```
}  // END populatebyrow
```

Example: mipex3.c

This example derives from `mipex1.c`. The differences between that simpler MIP example and this one are:

- The problem solved is slightly different so the output is interesting. The actual SOS and priority order that the example implements are arbitrary; they do not necessarily represent good data for this problem.
- The routine `CPXwriteprob()` writes the problem to disk before the example copies the SOS and priority order to verify that the base problem was copied correctly.
- The ILOG CPLEX preprocessing parameters for the presolver and aggregator are turned off to make the output interesting. Generally, we do not require nor recommend doing this.
- The routine `setsosandorder()` sets the SOS and priority order:
  - It calls `CPXcopysos()` to copy the SOS into the problem object.
  - It calls `CPXcopyorder()` to copy the priority order into the problem object.
  - It writes the SOS information to files by calling `CPXsoswrite()`.
  - It writes the priority order to files by calling `CPXordwrite()`.

```
#include <ilcplex/cplex.h>
#include <stdlib.h>

/* Bring in the declarations for the string functions */

#include <string.h>

/* Include declaration for function at end of program */

#ifndef  CPX_PROTOTYPE_MIN

static int
   setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                   int *objsen_p, double **obj_p, double **rhs_p,
                   char **sense_p, int **matbeg_p, int **matcnt_p,
                   int **matind_p, double **matval_p,
                   double **lb_p, double **ub_p, char **ctype_p),
   setsosandorder (CPXENVptr env, CPXLPptr lp);

static void
   free_and_null (char **ptr);


#else
```

```
static int
   setproblemdata (),
   setsosandorder ();

static void
   free_and_null ();

#endif


/* The problem we are optimizing will have 2 rows, 3 columns
   and 6 nonzeros.  */

#define NUMROWS    3
#define NUMCOLS    4
#define NUMNZ      9

#ifndef  CPX_PROTOTYPE_MIN
int
main (void)
#else
int
main ()
#endif
{
   /* Declare and allocate space for the variables and arrays that
      will contain the data which define the LP problem */

   char     *probname = NULL;
   int      numcols;
   int      numrows;
   int      objsen;
   double   *obj = NULL;
   double   *rhs = NULL;
   char     *sense = NULL;
   int      *matbeg = NULL;
   int      *matcnt = NULL;
   int      *matind = NULL;
   double   *matval = NULL;
   double   *lb = NULL;
   double   *ub = NULL;
   char     *ctype = NULL;

   /* Declare and allocate space for the variables and arrays where we
      will store the optimization results including the status, objective
      value, variable values, and row slacks. */

   int      solstat;
   double   objval;
   double   x[NUMCOLS];
   double   slack[NUMROWS];
```

```
CPXENVptr     env = NULL;
CPXLPptr      lp = NULL;
int           status;
int           i, j;
int           cur_numrows, cur_numcols;

/* Initialize the CPLEX environment */

env = CPXopenCPLEX (&status);

/* If an error occurs, the status value indicates the reason for
   failure.  A call to CPXgeterrorstring will produce the text of
   the error message.  Note that CPXopenCPLEX produces no output,
   so the only way to see the cause of the error is to use
   CPXgeterrorstring.  For other CPLEX routines, the errors will
   be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

if ( env == NULL ) {
   char  errmsg[1024];
   fprintf (stderr, "Could not open CPLEX environment.\n");
   CPXgeterrorstring (env, status, errmsg);
   fprintf (stderr, "%s", errmsg);
   goto TERMINATE;
}

/* Turn on output to the screen */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
if ( status ) {
   fprintf (stderr,
            "Failure to turn on screen indicator, error %d.\n", status);
   goto TERMINATE;
}

/* Fill in the data for the problem.  */

status = setproblemdata (&probname, &numcols, &numrows, &objsen, &obj,
                         &rhs, &sense, &matbeg, &matcnt, &matind, &matval,
                         &lb, &ub, &ctype);
if ( status ) {
   fprintf (stderr, "Failed to build problem data arrays.\n");
   goto TERMINATE;
}

/* Create the problem. */

lp = CPXcreateprob (env, &status, probname);

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
```

```
      channel from inside CPLEX.  In this example, the setting of
      the parameter CPX_PARAM_SCRIND causes the error message to
      appear on stdout.  */

   if ( lp == NULL ) {
      fprintf (stderr, "Failed to create LP.\n");
      goto TERMINATE;
   }

   /* Now copy the problem data into the lp */

   status = CPXcopylp (env, lp, numcols, numrows, objsen, obj, rhs,
                       sense, matbeg, matcnt, matind, matval,
                       lb, ub, NULL);

   if ( status ) {
      fprintf (stderr, "Failed to copy problem data.\n");
      goto TERMINATE;
   }

   /* Now copy the ctype array */

   status = CPXcopyctype (env, lp, ctype);
   if ( status ) {
      fprintf (stderr, "Failed to copy ctype\n");
      goto TERMINATE;
   }

   /* Write a copy of the problem to a file. */

   status = CPXwriteprob (env, lp, "mipex3.mps", NULL);
   if ( status ) {
      fprintf (stderr, "Failed to write LP to disk.\n");
      goto TERMINATE;
   }

   /* Set up the SOS set and priority order */

   status = setsosandorder (env, lp);
   if ( status ) goto TERMINATE;

   /* Turn off CPLEX presolve, aggregate, and print every node.  This
      is just to make it interesting.  Turning off CPLEX presolve is
      NOT recommended practice !! */

   status = CPXsetintparam (env, CPX_PARAM_PREIND, CPX_OFF);
   if (!status) CPXsetintparam (env, CPX_PARAM_AGGIND, CPX_OFF);
   if (!status) CPXsetintparam (env, CPX_PARAM_MIPINTERVAL, 1);

   if ( status ) {
      fprintf (stderr, "Failed to set some CPLEX parameters.\n");
      goto TERMINATE;
   }
   /* Optimize the problem and obtain solution. */

   status = CPXmipopt (env, lp);
   if ( status ) {
      fprintf (stderr, "Failed to optimize MIP.\n");
      goto TERMINATE;
   }

   solstat = CPXgetstat (env, lp);

   /* Write the output to the screen. */

   printf ("\nSolution status = %d\n", solstat);

   status = CPXgetmipobjval (env, lp, &objval);
   if ( status ) {
      fprintf (stderr,"No MIP objective value available.  Exiting...\n");
      goto TERMINATE;
   }

   printf ("Solution value  = %f\n\n", objval);

   /* The size of the problem should be obtained by asking CPLEX what
      the actual size is, rather than using what was passed to CPXcopylp.
      cur_numrows and cur_numcols store the current number of rows and
      columns, respectively.  */

   cur_numrows = CPXgetnumrows (env, lp);
   cur_numcols = CPXgetnumcols (env, lp);

   status = CPXgetmipx (env, lp, x, 0, cur_numcols-1);
   if ( status ) {
      fprintf (stderr, "Failed to get optimal integer x.\n");
      goto TERMINATE;
   }

   status = CPXgetmipslack (env, lp, slack, 0, cur_numrows-1);
   if ( status ) {
      fprintf (stderr, "Failed to get optimal slack values.\n");
      goto TERMINATE;
   }

   for (i = 0; i < cur_numrows; i++) {
      printf ("Row %d:  Slack = %10f\n", i, slack[i]);
   }

   for (j = 0; j < cur_numcols; j++) {
      printf ("Column %d:  Value = %10f\n", j, x[j]);
   }

TERMINATE:
   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   /* Free up the problem data arrays, if necessary. */

   free_and_null ((char **) &probname);
   free_and_null ((char **) &obj);
   free_and_null ((char **) &rhs);
   free_and_null ((char **) &sense);
   free_and_null ((char **) &matbeg);
   free_and_null ((char **) &matcnt);
   free_and_null ((char **) &matind);
   free_and_null ((char **) &matval);
   free_and_null ((char **) &lb);
   free_and_null ((char **) &ub);
   free_and_null ((char **) &ctype);

   return (status);

}  /* END main */


/* This function fills in the data structures for the mixed integer program:

      Maximize
       obj: x1 + 2 x2 + 3 x3 + x4
      Subject To
       c1: - x1 + x2 + x3 + 10x4  <= 20
       c2: x1 - 3 x2 + x3         <= 30
       c3:       x2       - 3.5x4  = 0
      Bounds
       0 <= x1 <= 40
       2 <= x4 <= 3
      Integers
        x2 x3 x4
      End
 */


#ifndef  CPX_PROTOTYPE_MIN
static int
setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                int *objsen_p, double **obj_p, double **rhs_p,
                char **sense_p, int **matbeg_p, int **matcnt_p,
                int **matind_p, double **matval_p,
                double **lb_p, double **ub_p, char **ctype_p)
#else
static int
setproblemdata (probname_p, numcols_p, numrows_p, objsen_p, obj_p,
                rhs_p, sense_p, matbeg_p, matcnt_p, matind_p, matval_p,
                lb_p, ub_p, ctype_p)
char    **probname_p;
int     *numcols_p;
int     *numrows_p;
int     *objsen_p;
double  **obj_p;
double  **rhs_p;
char    **sense_p;
int     **matbeg_p;
int     **matcnt_p;
int     **matind_p;
double  **matval_p;
double  **lb_p;
double  **ub_p;
char    **ctype_p;
#endif
{
   char     *zprobname = NULL;     /* Problem name <= 16 characters */
   double   *zobj = NULL;
   double   *zrhs = NULL;
   char     *zsense = NULL;
   int      *zmatbeg = NULL;
   int      *zmatcnt = NULL;
   int      *zmatind = NULL;
   double   *zmatval = NULL;
   double   *zlb = NULL;
   double   *zub = NULL;
   char     *zctype = NULL;
   int      status = 0;
```

```
zprobname = (char *) malloc (16 * sizeof(char));
zobj      = (double *) malloc (NUMCOLS * sizeof(double));
zrhs      = (double *) malloc (NUMROWS * sizeof(double));
zsense    = (char *) malloc (NUMROWS * sizeof(char));
zmatbeg   = (int *) malloc (NUMCOLS * sizeof(int));
zmatcnt   = (int *) malloc (NUMCOLS * sizeof(int));
zmatind   = (int *) malloc (NUMNZ * sizeof(int));
zmatval   = (double *) malloc (NUMNZ * sizeof(double));
zlb       = (double *) malloc (NUMCOLS * sizeof(double));
zub       = (double *) malloc (NUMCOLS * sizeof(double));
zctype    = (char *) malloc (NUMCOLS * sizeof(char));

if ( zprobname == NULL || zobj    == NULL ||
     zrhs      == NULL || zsense  == NULL ||
     zmatbeg   == NULL || zmatcnt == NULL ||
     zmatind   == NULL || zmatval == NULL ||
     zlb       == NULL || zub     == NULL ||
     zctype    == NULL                       )  {
   status = 1;
   goto TERMINATE;
}

strcpy (zprobname, "example");

/* The code is formatted to make a visual correspondence
   between the mathematical linear program and the specific data
   items.   */

  zobj[0] = 1.0;   zobj[1]   = 2.0;  zobj[2] = 3.0;    zobj[3] = 1.0;

zmatbeg[0] = 0;    zmatbeg[1] = 2;   zmatbeg[2] = 5;   zmatbeg[3] = 7;
zmatcnt[0] = 2;    zmatcnt[1] = 3;   zmatcnt[2] = 2;   zmatcnt[3] = 2;

zmatind[0] = 0;    zmatind[2] = 0;   zmatind[5] = 0;   zmatind[7] = 0;
zmatval[0] = -1.0; zmatval[2] = 1.0; zmatval[5] = 1.0; zmatval[7] = 10.0;

zmatind[1] = 1;    zmatind[3] = 1;    zmatind[6] = 1;
zmatval[1] = 1.0;  zmatval[3] = -3.0; zmatval[6] = 1.0;

                   zmatind[4] = 2;                     zmatind[8] = 2;
                   zmatval[4] = 1.0;                   zmatval[8] = -3.5;

zlb[0] = 0.0;    zlb[1] = 0.0;          zlb[2] = 0.0;          zlb[3] = 2.0;
zub[0] = 40.0;   zub[1] = CPX_INFBOUND; zub[2] = CPX_INFBOUND; zub[3] = 3.0;

zctype[0] = ’C’;   zctype[1] = ’I’;  zctype[2] = ’I’;   zctype[3] = ’I’;

/* The right-hand-side values don’t fit nicely on a line above.  So put
   them here.  */

zsense[0] = ’L’;
zrhs[0]   = 20.0;
```

```
   zsense[1] = ’L’;
   zrhs[1]   = 30.0;

   zsense[2] = ’E’;
   zrhs[2]   = 0.0;

TERMINATE:

   if ( status ) {
      free_and_null ((char **) &zprobname);
      free_and_null ((char **) &zobj);
      free_and_null ((char **) &zrhs);
      free_and_null ((char **) &zsense);
      free_and_null ((char **) &zmatbeg);
      free_and_null ((char **) &zmatcnt);
      free_and_null ((char **) &zmatind);
      free_and_null ((char **) &zmatval);
      free_and_null ((char **) &zlb);
      free_and_null ((char **) &zub);
      free_and_null ((char **) &zctype);
   }
   else {
      *numcols_p   = NUMCOLS;
      *numrows_p   = NUMROWS;
      *objsen_p    = CPX_MAX;   /* The problem is maximization */

      *probname_p  = zprobname;
      *obj_p       = zobj;
      *rhs_p       = zrhs;
      *sense_p     = zsense;
      *matbeg_p    = zmatbeg;
      *matcnt_p    = zmatcnt;
      *matind_p    = zmatind;
      *matval_p    = zmatval;
      *lb_p        = zlb;
      *ub_p        = zub;
      *ctype_p     = zctype;
   }
   return (status);

}  /* END setproblemdata */


#ifndef  CPX_PROTOTYPE_MIN
static int
setsosandorder (CPXENVptr env, CPXLPptr lp)
#else
static int
setsosandorder (env, lp)
CPXENVptr  env;
CPXLPptr   lp;
#endif
```

```
{
   /* Priority order information */
   int  colindex[2];
   int  priority[2];
   int  direction[2];

   /* SOS set information */
   char   sostype[1];
   int    sospri[1];
   int    sosbeg[1];
   int    sosind[2];
   double sosref[2];

   int  status = 0;

   /* Note - for this example, the priority order and SOS information
      are just made up for illustrative purposes.  The priority order
      is not necessarily a good one for this particular problem.  */

   /* Set order info.  Variables 1 and 3 will be in the priority order,
      with respective priorities of 8 and 7, and with respective
      branching directions of up and down */

   colindex[0]  = 1;                 colindex[1]  = 3;
   priority[0]  = 8;                 priority[1]  = 7;
   direction[0] = CPX_BRANCH_UP;     direction[1] = CPX_BRANCH_DOWN;

   status = CPXcopyorder (env, lp, 2, colindex, priority, direction);
   if ( status ) {
      fprintf (stderr, "CPXcopyorder failed.\n");
      goto TERMINATE;
   }

   /* Set SOS set info.  Create one SOS type 1 set, with variables
      2 and 3 in it, with set priority 3, and reference values
      25 and 18  for the 2 variables, respectively.   */

   sostype[0] = CPX_TYPE_SOS1;
   sospri[0]  = 3;
   sosbeg[0]  = 0;
   sosind[0]  = 2;    sosind[1] = 3;
   sosref[0]  = 25;   sosref[1] = 18;

   status = CPXcopysos (env, lp, 1, 2, sostype, sospri,
                        sosbeg, sosind, sosref);
   if ( status ) {
      fprintf (stderr, "CPXcopysos failed.\n");
      goto TERMINATE;
   }

   /* To assist in debugging, write the order and SOS to a file. */

   status = CPXordwrite (env, lp, "mipex3.ord");
   if ( status ) {
      fprintf (stderr, "CPXordwrite failed.\n");
      goto TERMINATE;
   }

   status = CPXsoswrite (env, lp, "mipex3.sos");
   if ( status ) {
      fprintf (stderr, "CPXsoswrite failed.\n");
      goto TERMINATE;
   }

TERMINATE:

   return (status);

}




/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

#ifndef  CPX_PROTOTYPE_MIN
static void
free_and_null (char **ptr)
#else
static void
free_and_null (ptr)
char  **ptr;
#endif
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */
```

CHAPTER

6

Solving Network-Flow Problems

This chapter tells you more about the ILOG CPLEX Network Optimizer. It includes sections on:

Choosing an Optimizer: Network Considerations

As we explain in *Using the Callable Library in an Application* on page 57, to exploit ILOG CPLEX in your own application, you must first create a ILOG CPLEX environment, instantiate a problem object, and populate the problem object with data. As your next step, you call a ILOG CPLEX optimizer.

If part of your problem is structured as a network, then you may want to consider calling the ILOG CPLEX Network Optimizer. This optimizer may have a positive impact on performance. There are two alternative ways of calling the network optimizer:

- If your entire problem consists of a network flow, you should consider creating a *network object* instead of an LP object. Then populate it, and solve it with the network optimizer. This alternative generally yields the best performance because it does not incur the overhead of LP data structures.
- If your problem is an LP where a large part is a network structure, you may call the network optimizer for the populated LP object.

How much performance improvement you observe between using only a simplex optimizer versus using the network optimizer followed by either of the simplex optimizers depends on the number and nature of the other constraints in your problem. On a pure network problem, we have measured performance 100 times faster with the network optimizer. However, if the network component of your problem is small relative to its other parts, then using the solution of the network part of the problem as a starting point for the remainder may or may not improve performance, compared to running the primal or dual simplex optimizer. Only experiments with your own problem can tell.

Formulating a Network Problem

A *network-flow* problem finds the minimal-cost flow through a *network*, where a network consists of a set N of nodes and a set A of arcs connecting the nodes. An arc a in the set A is an ordered pair (i, j) where i and j are nodes in the set N; node i is called the *tail* or the from-node and node j is called the *head* or the to-node of the arc a. Not all the pairs of nodes in a set N are necessarily connected by arcs in the set A. More than one arc may connect a pair of nodes; in other words, $a_1 = (i, j)$ and $a_2 = (i, j)$ may be two arcs in A, both connecting the nodes i and j in N.

Each arc may be associated with four values:

- x_a is the flow value, that is, the amount passing through the arc a from its tail (or from-node) to its head (or to-node). The flow values are the modeling variables of a network-flow problem. Negative values are allowed; a negative flow value indicates that there is flow from the head to the tail.
- l_a, the lower bound, determines the minimum flow allowed through the arc a. By default, the lower bound on an arc is 0 (zero).
- u_a, the upper bound, determines the maximum flow allowed through the arc a. By default, the upper bound on an arc is positive infinity.
- c_a, the objective value, determines the contribution to the objective function of one unit of flow through the arc.

Each node is associated with one value:

- s_n is the supply value at node n.

By convention, a node with strictly positive supply value (that is, $s_n > 0$) is called a *supply* node or a *source*, and a node with strictly negative supply value (that is, $s_n < 0$) is called a *demand* node or a *sink*. A node where $s_n = 0$ is called a *transshipment* node. The sum of all supplies must match the sum of all demands; if not, then the network flow problem is *infeasible*.

T_n is the set of arcs whose tails are node *n*; H_n is the set of arcs whose heads are node *n*. The usual form of a network problem looks like this:

Minimize (or maximize) $\sum_{a \in A} (c_a x_a)$

subject to $\sum_{a \in T_n} x_a - \sum_{a \in H_n} x_a = s_n \forall (n \in N)$

with these bounds $l_a \le x_a \le u_a \forall (a \in A)$

That is, for each node, the net flow entering and leaving the node must equal its supply value, and all flow values must be within their bounds. The solution of a network-flow problem is an assignment of flow values to arcs (that is, the modeling variables) to satisfy the problem formulation. A flow that satisfies the constraints and bounds is *feasible*.

Example: Network Optimizer in the Interactive Optimizer

This example is based on a network where the aim is to minimize cost and where the flow through the network has both cost and capacity. Figure 6.1 shows you the nodes and arcs of this network. The nodes are labeled by their identifying node number from 1 through 8. The number inside a node indicates its supply value; 0 (zero) is assumed where no number is given. The arcs are labeled 1 through 14. The lower bound `l`, upper bound `u`, and objective value `c` of each arc are displayed in parentheses `(l, u, c)` beside each arc. In this example, node 1 and node 5 are sources, representing a positive net flow, whereas node 4 and node 8 are sinks, representing negative net flow.

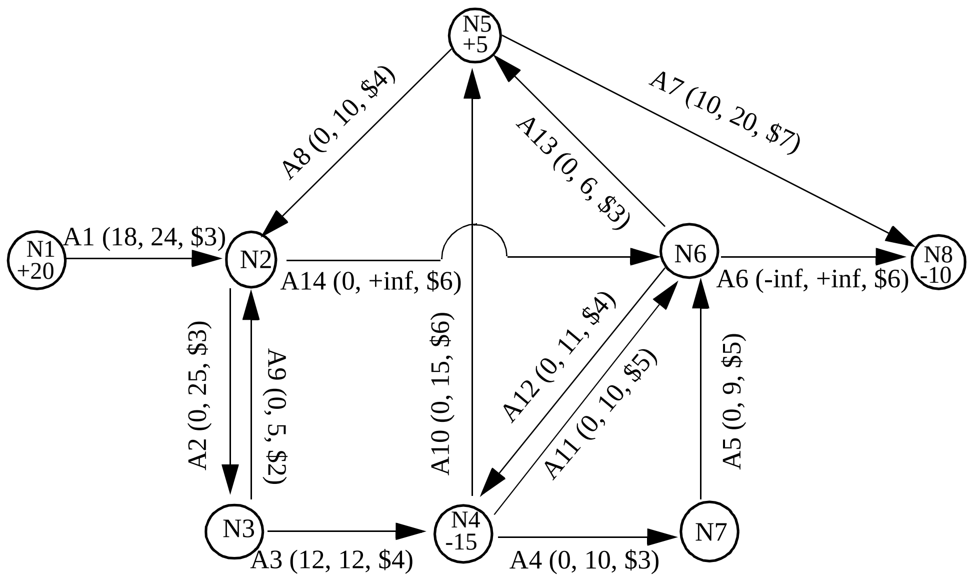

Figure 6.1 *A Directed Network with Arc-Capacity, Flow-Cost, Sinks, and Sources*

In the standard distribution of ILOG CPLEX, the file `nexample.net` contains the formatted problem formulation for this example. You can read it into the Interactive Optimizer with the command `read nexample.net`. After you read the problem into the Interactive Optimizer, you can solve it with the command `netopt` or the command `optimize`.

Understanding the Network Log File

As ILOG CPLEX solves the problem, it produces a log like the following lines:

```
Iteration log . . .
Iteration:     0   Infeasibility     =           48.000000 (5.15396e+13)

Network - Optimal:  Objective =    2.6900000000e+02
Solution time =    0.00 sec.  Iterations = 7 (7)
```

This network log file differs slightly from the log files produced by other ILOG CPLEX optimizers: it contains values enclosed in parentheses that represent modified objective function values.

As long as the network optimizer has not yet found a feasible solution, we say that it is in Phase I. In Phase I, the network optimizer uses modified objective coefficients that penalize infeasibility. At its default settings, the ILOG CPLEX Network Optimizer displays the value of the objective function calculated in terms of these modified objective coefficients in parentheses in the network log file.

You can control the amount of information recorded in the network log file, just as you control the amount of information in other ILOG CPLEX log files. To record no information at all in the log file, use the command `set network display 0`. To display the current objective value in parentheses relative to the actual unmodified objective coefficients, use the command `set network display 1`. To see the display we described earlier in this section, leave the network display parameter at its default value, `2`. (If you have changed the default value, you can reset it with the command `set network display 2`.)

Tuning Performance of the Network Optimizer

The default values of parameters controlling the network optimizer are generally the best choices for effective performance. However, the following sections indicate parameters that you may want to experiment with in your particular problem.

Controlling Tolerance

You control the feasibility tolerance for the network optimizer through the feasibility tolerance parameter. In the Interactive Optimizer, use the command
`set network tolerances feasibility`.

Likewise, you control the optimality tolerance for the network optimizer through the optimality tolerance parameter. Table 6.1 and Table 6.2 summarize the default value, range, and parameter name.

Table 6.1 *Network Tolerance Parameter: Optimality*

	Optimality tolerance
Default Value	$1e^{-6}$
Range	$0.1 - 1e^{-11}$
Callable Parameter	`CPX_PARAM_NETEPOPT`
Interactive Option	`network tolerances optimality`

Table 6.2 *Network Tolerance Parameter: Feasibility*

	Feasibility tolerance
Default Value	$1e^{-6}$
Range	$0.1 - 1e^{-11}$
Callable Parameter	`CPX_PARAM_NETEPRHS`
Interactive Option	`network tolerances feasibility`

Selecting a Pricing Algorithm for the Network Optimizer

On the rare occasions when the network optimizer seems to take too long to find a solution, you may want to change the pricing algorithm to try to speed up computation. In the Interactive Optimizer, use the command `set network pricing i`, substituting a value for `i` to indicate which pricing algorithm to use. All the choices use variations of partial reduced-cost pricing.

Limiting Iterations in the Network Optimizer

Use the command `set network iterations i`, substituting a value for `i`, if you want to limit the number of iterations that the network optimizer performs.

Changing Sense: from Min to Max

To change a minimization problem to a maximization problem in the Interactive Optimizer, use the command `change sense max` and optimize again. For example, here is a transcript of a session in the Interactive Optimizer where we have already entered `nexample.net` and optimized it, and we now change its sense and optimize again:

```
CPLEX> change sense max
Problem is a minimization problem.
Problem is now a maximization problem.
CPLEX> netopt
Iteration log . . . Iteration: 0  Objective =         269.000000
Network - Optimal:  Objective =    5.0400000000e+02
Solution time =    0.00 sec.  Iterations = 5 (0)
```

Because we had already solved this example once as a minimization problem, the maximization started from a feasible solution. You control whether or not the network optimizer starts from an existing solution: use the command `set advance 1` to indicate in the Interactive Optimizer that you want to start from an advanced basis. This setting is the default.

Example: Using the Network Optimizer with the Callable Library

In the standard distribution of ILOG CPLEX, the file `netex1.c` contains code that creates, solves, and displays the solution of the network-flow problem illustrated in Figure 6.1 on page 222.

Briefly, the `main()` function initializes the ILOG CPLEX environment and creates the problem object; it also calls the optimizer to solve the problem and retrieves the solution.

In detail, `main()` first calls the Callable Library routine `CPXopenCPLEX()`. As we explain in *Initialize the ILOG CPLEX Environment* on page 57, `CPXopenCPLEX()` must always be the first ILOG CPLEX routine called in a ILOG CPLEX Callable Library application. Those routines create the ILOG CPLEX environment and return a pointer (called `env`) to it. This pointer will be passed to every Callable Library routine. If this initialization routine fails, env will be NULL and the error code indicating the reason for the failure will be written to status. That error code can be transformed into a string by the Callable Library routine CPXgeterrorstring().

After main() initializes the ILOG CPLEX environment, it uses the Callable Library routine CPXsetintparam() to turn on the ILOG CPLEX screen indicator parameter CPX_PARAM_SCRIND so that ILOG CPLEX output appears on screen. If this parameter is turned off, ILOG CPLEX does not produce viewable output, neither on screen, nor in a log file. We recommend turning this parameter on when you are debugging your application.

The Callable Library routine CPXNETcreateprob() creates an *empty* problem object, that is, a minimum-cost network-flow problem with no arcs and no nodes.

The function buildNetwork() populates the problem object; that is, it loads the problem data into the problem object. Pointer variables in the example are initialized as NULL so that you can check whether they point to valid data—a good programming practice. The most important calls in this function are to the Callable Library routines, CPXNETaddnodes(), which adds nodes with the specified supply values to the network problem, and CPXNETaddarcs(), which adds the arcs connecting the nodes with the specified objective values and bounds. In this example, both routines are called with their last argument NULL indicating that no names are assigned to the network nodes and arcs. If you want to name arcs and nodes in your problem, pass an array of strings instead.

The function buildNetwork() also includes a few routines that are not strictly necessary to this example, but illustrate concepts you may find useful in other applications. To delete a node and all arcs dependent on that node, it uses the Callable Library routine CPXNETdelnodes(). To change the objective sense to minimization, it uses the Callable Library routine CPXNETchgobjsen().

Also buildNetwork() sets the row growth and column growth parameters CPX_PARAM_ROWGROWTH and CPX_PARAM_COLGROWTH. These parameters specify the amount that internal arrays are extended if more nodes or arcs are added than currently fit in allocated memory. If you build up a problem by adding nodes and arcs one by one, and if these parameters are set to a low value, then internal arrays will be frequently reallocated; frequent reallocation may negatively impact performance. Ideally, these parameters are set to the maximum number of nodes and arcs that the problem will ever have. This setting will avoid all reallocation and therefore provide best performance. The parameter CPX_PARAM_ROWGROWTH pertains to adding nodes to a network problem (and rows to an LP, QP, or MIP problem) whereas CPX_PARAM_COLGROWTH pertains to adding arcs to a network problem (or columns to an LP, QP, or MIP problem).

Let's return to main(), where it actually calls the network optimizer with the Callable Library routine, CPXNETprimopt(). For CPXNETprimopt(), the return value 0 means that solution information is available in the network object. Before retrieving that solution, we allocate arrays to hold it. Then we use CPXNETsolution() to copy the solution into those arrays. After we display the solution on screen, we write the network problem into a file, netex1.net in the NET file format.

The TERMINATE: label is used as a place for the program to exit if any type of error occurs. Therefore, code following this label cleans up: it frees the memory that has been allocated for the solution data; it frees the network object by calling CPXNETfreeprob(); and it frees the ILOG CPLEX environment by calling CPXcloseCPLEX(). All freeing should be done only if the data is actually available. The Callable Library routine CPXcloseCPLEX() should always be the last ILOG CPLEX routine called in a ILOG CPLEX Callable Library application. In other words, all ILOG CPLEX objects that have been allocated should be freed before the call to CPXcloseCPLEX().

Complete Program: netex1.c

The complete program, netex1.c, appears here or online in the standard distribution.

```
#include <ilcplex/cplex.h>
#include <stdlib.h>

/* Import the declarations for the string functions */

#include <string.h>



/* Forward declaration for function at end of program */

#ifndef  CPX_PROTOTYPE_MIN

static int
   buildNetwork  (CPXENVptr env, CPXNETptr net);

static void
   free_and_null (char **ptr);

#else

static int
   buildNetwork  ();

static void
   free_and_null ();

#endif


#ifndef  CPX_PROTOTYPE_MIN
int
main (void)
#else
int
main ()
#endif
{
   /* Declare variables and arrays for retrieving problem data and
      solution information later on. */

   int      narcs;
   int      nnodes;
   int      solstat;
   double   objval;
   double   *x     = NULL;
   double   *pi    = NULL;
   double   *slack = NULL;
   double   *dj    = NULL;

   CPXENVptr env = NULL;
   CPXNETptr net = NULL;
   int       status;
   int       i, j;

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no
      output, so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
      be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

   if ( env == NULL ) {
      char  errmsg[1024];
      fprintf (stderr, “Could not open CPLEX environment.\n”);
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, “%s”, errmsg);
      goto TERMINATE;
   }

   /* Turn on output to the screen */

   status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
   if ( status ) {
      fprintf (stderr,
               “Failure to turn on screen indicator, error %d.\n”, status);
      goto TERMINATE;
   }

   /* Create the problem. */

   net = CPXNETcreateprob (env, &status, “netex1”);

   /* A returned pointer of NULL may mean that not enough memory
      was available or there was some other problem.  In the case of
      failure, an error message will have been written to the error
      channel from inside CPLEX.  In this example, the setting of
      the parameter CPX_PARAM_SCRIND causes the error message to
      appear on stdout.  */

   if ( net == NULL ) {
      fprintf (stderr, “Failed to create network object.\n”);
      goto TERMINATE;
   }

   /* Fill in the data for the problem.  Note that since the space for
```

```
      the data already exists in local variables, we pass the arrays
      directly to the routine to fill in the data structures.  */

   status = buildNetwork (env, net);

   if ( status ) {
      fprintf (stderr, “Failed to build network problem.\n”);
      goto TERMINATE;
   }

   /* Optimize the problem and obtain solution. */

   status = CPXNETprimopt (env, net);
   if ( status ) {
      fprintf (stderr, “Failed to optimize network.\n”);
      goto TERMINATE;
   }

   /* get network dimensions */

   narcs  = CPXNETgetnumarcs  (env, net);
   nnodes = CPXNETgetnumnodes (env, net);

   /* allocate memory for solution data */

   x     = (double *) malloc (narcs  * sizeof (double));
   dj    = (double *) malloc (narcs  * sizeof (double));
   pi    = (double *) malloc (nnodes * sizeof (double));
   slack = (double *) malloc (nnodes * sizeof (double));

   if ( x     == NULL ||
        dj    == NULL ||
        pi    == NULL ||
        slack == NULL   ) {
      fprintf (stderr, “Failed to allocate arrays.\n”);
      goto TERMINATE;
   }

   status = CPXNETsolution (env, net, &solstat, &objval, x, pi, slack, dj);
   if ( status ) {
      fprintf (stderr, “Failed to obtain solution.\n”);
      goto TERMINATE;
   }

   /* Write the output to the screen. */

   printf (“\nSolution status = %d\n”, solstat);
   printf (“Solution value  = %f\n\n”, objval);

   for (i = 0; i < nnodes; i++) {
      printf (“Node %2d:  Slack = %10f  Pi = %10f\n”, i, slack[i], pi[i]);
   }

   for (j = 0; j < narcs; j++) {
      printf (“Arc  %2d:  Value = %10f  Reduced cost = %10f\n”,
              j, x[j], dj[j]);
   }
   /* Finally, write a copy of the problem to a file. */

   status = CPXNETwriteprob (env, net, “netex1.net”, NULL);
   if ( status ) {
      fprintf (stderr, “Failed to write network to disk.\n”);
      goto TERMINATE;
   }

TERMINATE:

   /* Free memory for solution data */

   free_and_null ((char **) &x);
   free_and_null ((char **) &dj);
   free_and_null ((char **) &pi);
   free_and_null ((char **) &slack);

   /* Free up the problem as allocated by CPXNETcreateprob, if necessary */

   if ( net != NULL ) {
      CPXNETfreeprob (env, &net);
      if ( status ) {
         fprintf (stderr, “CPXNETfreeprob failed, error code %d.\n”, status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
      char  errmsg[1024];
         fprintf (stderr, “Could not close CPLEX environment.\n”);
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, “%s”, errmsg);
      }
   }

   return (status);

}  /* END main */



#ifndef  CPX_PROTOTYPE_MIN
static int
buildNetwork (CPXENVptr env, CPXNETptr net)
#else
static int
buildNetwork (env, net)
CPXENVptr env;
CPXNETptr net;
```

```
#endif
{
   int status = 0;

   /* definitions to improve readability */

#  define NNODES  8
#  define NARCS  14
#  define inf    CPX_INFBOUND

   /* Define list of supply values for the nodes */

   double supply[NNODES] = {20.0, 0.0, 0.0, -15.0, 5.0, 0.0, 0.0, -10.0};

   /* Define list of tail or from-node indices as well as head or
      to-node indices for the arcs.  Notice that according to C
      standard the first node has index 0. */

   int    tail[NARCS] = {   0,    1,    2,    3,    6,    5,    4,
                            4,    2,    3,    3,    5,    5,    1};
   int    head[NARCS] = {   1,    2,    3,    6,    5,    7,    7,
                            1,    1,    4,    5,    3,    4,    5};

   /* Define list of objective values and lower and upper bound values
      for the arcs */

   double obj [NARCS] = { 3.0,  3.0,  4.0,  3.0,  5.0,  6.0,  7.0,
                          4.0,  2.0,  6.0,  5.0,  4.0,  3.0,  6.0};
   double ub  [NARCS] = {24.0, 25.0, 12.0, 10.0,  9.0,  inf, 20.0,
                         10.0,  5.0, 15.0, 10.0, 11.0,  6.0,  inf};
   double lb  [NARCS] = {18.0,  0.0, 12.0,  0.0,  0.0, -inf,  0.0,
                          0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0};

   /* Delete existing network.  This is not necessary in this
      context since we know we have an empty network object.
      Notice that CPXNETdelnodes deletes all arcs incident to
      the deleted nodes as well.  Therefore this one function
      call effectively deletes an existing network problem. */

   if ( CPXNETgetnumnodes (env, net) > 0 ) {
      status = CPXNETdelnodes (env, net, 0,
                               CPXNETgetnumnodes (env, net)-1);
      if ( status ) goto TERMINATE;
   }

   /* Set growth rates for rows/nodes and columns/arcs.  This
      is to avoid internal memory reallocations while adding
      nodes and arcs.  Since we are adding all nodes and all
      arcs using only one function call for each it is actually
      unnecessary, but if more function calls are used, finding
      the right settings may improve performance. */

   status = CPXsetintparam (env, CPX_PARAM_ROWGROWTH, NNODES);
   if ( status ) goto TERMINATE;

   status = CPXsetintparam (env, CPX_PARAM_COLGROWTH, NARCS);
   if ( status ) goto TERMINATE;
   /* Set optimization sense */

   status = CPXNETchgobjsen (env, net, CPX_MIN);
   if ( status ) goto TERMINATE;

   /* Add nodes to network along with their supply values,
      but without any names. */

   status = CPXNETaddnodes (env, net, NNODES, supply, NULL);
   if ( status ) goto TERMINATE;

   /* Add arcs to network along with their objective values and
      bounds, but without any names. */

   status = CPXNETaddarcs (env, net, NARCS, tail, head, lb, ub, obj, NULL);
   if ( status ) goto TERMINATE;

TERMINATE:

   return (status);

}  /* END buildnetwork */


#ifndef  CPX_PROTOTYPE_MIN
static void
free_and_null (char **ptr)
#else
static void
free_and_null (ptr)
char  **ptr;
#endif
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */
```

Solving Network-Flow Problems as LP Problems

A network-flow model is an LP model with special structure. The ILOG CPLEX Network Optimizer is a highly efficient implementation of the primal simplex technique adapted to take advantage of this special structure. In particular, no basis factoring occurs. However, it is possible to solve network models using any of the ILOG CPLEX LP optimizers if first, you convert the network data structures to those of an LP model. To convert the network data structures to LP data structures, in the Interactive Optimizer, use the command `change problem lp`; from the Callable Library, use the routine `CPXcopynettolp()`.

The LP formulation of our example from Figure 6.1 on page 222 looks like this:

Minimize

$$3a_1 + 3a_2 + 4a_3 + 3a_4 + 5a_5 + 6a_6 + 7a_7 + 4a_8 + 2a_9 + 6a_{10} + 5a_{11} + 4a_{12} + 3a_{13} + 6a_{14}$$

subject to

$$\begin{array}{rrrrrrrrrrrrrrcr}
a_1 & & & & & & & & & & & & & & = & 20 \\
-a_1 & + a_2 & & & & & & - a_8 & - a_9 & & & & & + a_{14} & = & 0 \\
& - a_2 & + a_3 & & & & & & + a_9 & & & & & & = & 0 \\
& & - a_3 & + a_4 & & & & & & + a_{10} & + a11 & - a_{12} & & & = & -15 \\
& & & & & & a_7 & + a_8 & & - a_{10} & & & - a_{13} & & = & 5 \\
& & & & - a_5 & + a_6 & & & & & - a_{11} & + a_{12} & + a_{13} & - a_{14} & = & 0 \\
& & & - a_4 & + a_5 & & & & & & & & & & = & 0 \\
& & & & & - a_6 & - a_7 & & & & & & & & = & -10
\end{array}$$

with these bounds

$$\begin{array}{lllll}
18 & \leq & a_1 & \leq & 24 \\
0 & \leq & a_2 & \leq & 25 \\
 & & a_3 & = & 12 \\
0 & \leq & a_4 & \leq & 10 \\
0 & \leq & a_5 & \leq & 9 \\
 & & a_6 & \text{free} & \\
0 & \leq & a_7 & \leq & 20 \\
0 & \leq & a_8 & \leq & 10 \\
0 & \leq & a_9 & \leq & 5 \\
0 & \leq & a_{10} & \leq & 15 \\
0 & \leq & a_{11} & \leq & 10 \\
0 & \leq & a_{12} & \leq & 11 \\
0 & \leq & a_{13} & \leq & 6 \\
0 & \leq & a_{14} & &
\end{array}$$

In that formulation, in each column there is exactly one coefficient equal to *1* (one), exactly one coefficient equal to *-1*, and all other coefficients are *0* (zero).

Since a network-flow problem corresponds in this way to an LP problem, you can indeed solve a network-flow problem by means of a ILOG CPLEX LP optimizer as well. If you read a network-flow problem into the Interactive Optimizer, you can transform it into its LP formulation with the command `change problem lp`. After this change, you can apply any of the LP optimizers to this problem.

When you change a network-flow problem into an LP problem, the basis information that is available in the network-flow problem is passed along to the LP formulation. In fact, if you have already solved the network-flow problem to optimality, then if you call the primal or dual simplex optimizers (for example, with the Interactive Optimizer command `primopt` or `tranopt`), that simplex optimizer will perform no iterations.

Generally, you can also use the same basis from a basis file for both the LP and the network optimizers. However, there is one exception: in order to use an LP basis with the network optimizer, at least one slack variable or one artificial variable needs to be basic. *Starting from an Advanced Basis* on page 101 explains more about this topic in the context of LP optimizers.

If you have already read the LP formulation of a problem into the Interactive Optimizer, you can transform it into a network with the command `change problem network`. Given any LP problem and this command, ILOG CPLEX will try to find the largest network embedded in the LP problem and transform it into a network-flow problem. However, as it does so, it discards all rows and columns that are not part of the embedded network. At the same time, ILOG CPLEX passes along as much basis information as possible to the network optimizer.

Example: Network to LP Transformation

This example shows how to transform a network-flow problem into its corresponding LP formulation. That example also indicates why you might want to make such a change. The example reads a network-flow problem from a file (rather than populating the problem object by adding rows and columns as we did in `netex1.c`). It then attempts to solve the problem by calling the Callable Library routine `CPXNETprimopt()`. If it determines that the problem is infeasible, it then transforms the problem into its LP formulation so that the infeasibility finder can analyze the problem and possibly indicate the cause of the infeasibility in an irreducibly inconsistent set (IIS). To perform this analysis, the application calls the Callable Library routine `CPXiiswrite()` to write the IIS to the file `netex2.iis`.

Complete Program: netex2.c

The complete program, `netex2.c`, appears here or online in the standard distribution.

```
#include <ilcplex/cplex.h>
```

```
/* Import the declarations for the string functions */

#include <string.h>




#ifndef  CPX_PROTOTYPE_MIN
int
main (int argc, char **argv)
#else
int
main (argc, argv)
int  argc;
char **argv;
#endif
{
   /* Declare variables and arrays for retrieving problem data and
      solution information later on. */

   int       status = 0;
   CPXENVptr env = NULL;
   CPXNETptr net = NULL;
   CPXLPptr  lp  = NULL;

   /* Check command line */

   if ( argc != 2 ) {
      fprintf (stderr, “Usage: %s <network file>\n”, argv[0]);
      fprintf (stderr, “Exiting ...\n”);
      goto TERMINATE;
   }

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no
      output, so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
      be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

   if ( env == NULL ) {
      char  errmsg[1024];
      fprintf (stderr, “Could not open CPLEX environment.\n”);
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, “%s”, errmsg);
      goto TERMINATE;
   }

   /* Turn on output to the screen */

   status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
   if ( status ) {
      fprintf (stderr,
               “Failure to turn on screen indicator, error %d.\n”, status);
      goto TERMINATE;
   }

   /* Create the problem. */

   net = CPXNETcreateprob (env, &status, “netex2”);

   /* A returned pointer of NULL may mean that not enough memory
      was available or there was some other problem.  In the case of
      failure, an error message will have been written to the error
      channel from inside CPLEX.  In this example, the setting of
      the parameter CPX_PARAM_SCRIND causes the error message to
      appear on stdout.  */

   if ( net == NULL ) {
      fprintf (stderr, “Failed to create network object.\n”);
      goto TERMINATE;
   }

   /* Read network problem data from file
      with filename given as command line argument. */

   status = CPXNETreadcopyprob (env, net, argv[1]);

   if ( status ) {
      fprintf (stderr, “Failed to build network problem.\n”);
      goto TERMINATE;
   }

   /* Optimize the problem */

   status = CPXNETprimopt (env, net);
   if ( status ) {
      fprintf (stderr, “Failed to optimize network.\n”);
      goto TERMINATE;
   }

   /* Check network solution status */

   if ( CPXNETgetstat (env, net) == CPX_INFEASIBLE ) {

      /* Create LP object used for invoking infeasibility finder */

      lp = CPXcreateprob (env, &status, “netex2”);
      if ( lp == NULL ) {
         fprintf (stderr, “Failed to create LP object.\n”);
         goto TERMINATE;
      }

      /* Copy LP representation of network problem to lp object, along
         with the current basis available in the network object. */

      status = CPXcopynettolp (env, lp, net);
      if ( status ) {
         fprintf (stderr, “Failed to copy network as LP.\n”);
         goto TERMINATE;
      }
```

```
      /* Optimize the LP with primal to create an LP solution.  This
         optimization will start from the basis previously generated by
         CPXNETprimopt() as long as the advance indicator is switched
         on (its default).  */

      status = CPXsetintparam (env, CPX_PARAM_LPMETHOD, CPX_ALG_PRIMAL);
      if ( status ) {
         fprintf (stderr,
                  "Failure to set LP method, error %d.\n", status);
         goto TERMINATE;
      }

      status = CPXlpopt (env, lp);
      if ( status ) {
         fprintf (stderr, "Failed to optimize LP.\n");
         goto TERMINATE;
      }

      /* Find IIS and write it to a file */

      status = CPXiiswrite (env, lp, "netex2.iis");
      if ( status ) {
         fprintf (stderr, "Failed to find IIS or write IIS file\n");
         goto TERMINATE;
      }

      printf ("IIS written to file netex2.iis\n");
   }
   else {
      printf ("Network problem not proved to be infeasible\n");
   }

TERMINATE:

   /* Free up the problem as allocated by CPXNETcreateprob, if necessary */

   if ( net != NULL ) {
      CPXNETfreeprob (env, &net);
      if ( status ) {
         fprintf (stderr, "CPXNETfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);
      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   return (status);

}  /* END main */
```

Solving LPs with the Network Optimizer

If you tell ILOG CPLEX to apply the network optimizer to an LP problem—whether in the Interactive Optimizer with the command `netopt` or from the Callable Library with the routine `CPXhybnetopt()`—ILOG CPLEX performs a sequence of steps. It first searches for a part of the LP that conforms to network structure. Such a part is known as an *embedded* network. It then uses the network optimizer to solve that embedded network. Next, it uses the resulting basis to construct a starting basis for the full LP problem. Finally, it solves the LP problem with a simplex optimizer.

Network Extraction

The ILOG CPLEX network extractor searches an LP constraint matrix for a submatrix with the following characteristics:

- the coefficients of the submatrix are all *0* (zero), *1* (one), or *-1 (minus one)*;
- each variable appears in at most two rows with at most one coefficent of *+1* and at most one coefficient of *-1*.

ILOG CPLEX can perform different levels of extraction. The level it performs depends on the `netfind` parameter.

- When the `netfind` parameter is set to `1` (one), ILOG CPLEX extracts only the obvious network; it uses no scaling; it scans rows in their natural order; it stops extraction as soon as no more rows can be added to the network found so far.
- When the `netfind` parameter is set to `2`, the default setting, ILOG CPLEX also uses reflection scaling (that is, it multiplies rows by `-1`) in an attempt to extract a larger network.

- When the `netfind` parameter is set to `3`, ILOG CPLEX uses general scaling, rescaling both rows and columns, in an attempt to extract a larger network.

In terms of total solution time expended, it may or may not be advantageous to extract the largest possible network. Characteristics of your problem will determine the tradeoff between network size and the number of simplex iterations required to finish solving the model after solving the embedded network.

To set the `netfind` parameter:

- In the Interactive Optimizer, use the command `set network netfind i`, substituting a value for `i`.
- From the Callable Library, use the routine `CPXsetintparam()` with arguments to indicate the environment, the parameter `CPX_PARAM_NETFIND`, and a value.

(This parameter is the same one that you use when you transform an LP model to a network-flow model, as described in *Solving LPs with the Network Optimizer* on page 237.)

Even if your problem does not conform precisely to network conventions, the network optimizer may still be advantageous to use. When it is possible to transform the original statement of a linear program into network conventions by these algebraic operations:

- changing the signs of coefficients,
- multiplying constraints by constants,
- rescaling columns,
- adding or eliminating redundant relations,

then ILOG CPLEX will carry out such transformations automatically if you set the netfind parameter appropriately.

Preprocessing and the Network Optimizer

If your LP problem includes network structures, there is a possibility that ILOG CPLEX preprocessing may eliminate those structures from your model. For that reason, you should consider turning off preprocessing before you invoke the network optimizer on an LP problem.

C H A P T E R

7w

Solving Quadratic Programming Problems

This chapter tells you about solving *convex quadratic programming* problems (QPs) with the ILOG CPLEX Barrier Optimizer. (To use the ILOG CPLEX Barrier Optimizer in linear programs (LPs), see *Solving LP Problems with the Barrier Optimizer* on page 129.)

This chapter contains sections on:

- Identifying Convex Quadratic Programming Problems
- Entering QPs
- Saving QP Problems
- Changing Problem Type in QPs
- Changing Quadratic Terms
- Optimizing QPs with the Barrier Optimizer
- Example: Creating a QP, Optimizing, Finding a Solution
- Example: Reading a QP from a File

To use the ILOG CPLEX Barrier Optimizer in application development, you must hold a special, optional, development license. If you call barrier routines from the ILOG CPLEX Callable Library in your applications, your end user must be licensed for runtime or derived work. For more information about ILOG CPLEX licensing, contact your ILOG CPLEX representative.

Identifying Convex Quadratic Programming Problems

Conventionally, a quadratic program (QP) is formulated this way:

Minimize $^1/_2\, \mathbf{x}^T \mathbf{Q} x + \mathbf{c}^T \mathbf{x}$

subject to $\mathbf{Ax} \sim b$

with these bounds $l \leq \mathbf{x} \leq u$

where the relation ~ may be any combination of equal to, less than or equal to, greater than or equal to, or range constraints. As in other problem formulations, l indicates lower and u upper bounds. $\mathbf{Q}$ is a matrix of objective function coefficients. That is, the elements Q_{jj} are the coefficients of the quadratic terms x_j^2, and the elements Q_{ij} and Q_{ji} are summed together to be the coefficient of the term $x_i x_j$.

ILOG CPLEX distinguishes two kinds of $\mathbf{Q}$ matrices:

- In a *separable* problem, only the diagonal terms of the matrix are defined.
- In a *nonseparable* problem, at least one off-diagonal term of the matrix is nonzero.

ILOG CPLEX optimizes only *convex* quadratic *minimization* problems or equivalently, only *concave* quadratic *maximization* problems, as illustrated in Figure 7.1. For convex QPs, $\mathbf{Q}$ must be *positive semi-definite*; that is, the term $\mathbf{x'Qx} \geq 0$ for all $\mathbf{x}$, whether or not $\mathbf{x}$ is feasible. For concave maximization problems, the requirement is that $\mathbf{Q}$ must be *negative semi-definite*; that is, $\mathbf{x'Qx} \leq 0$ for all $\mathbf{x}$. In a minimization problem, if $\mathbf{Q}$ is separable and positive semi-definite, then $\mathbf{Q} \geq 0$.

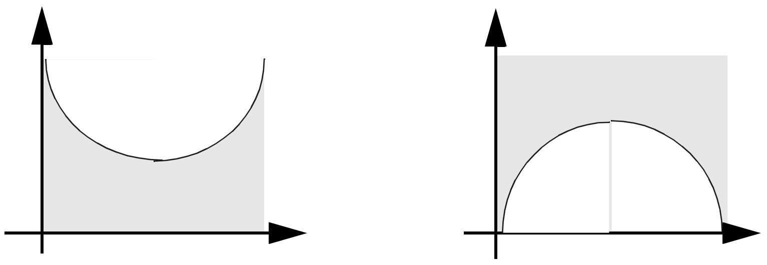

Figure 7.1 *Maximize a Concave Objective Function, Minimize a Convex Objective Function*

In this chapter, we assume you have some familiarity with quadratic programming. For a more complete explanation of quadratic programming generally, we recommend you consult a text such as one of those listed in *Further Reading* on page 25 of the preface of this manual.

Entering QPs

As you see in the problem formulation, ILOG CPLEX assumes there is an initial factor, *1/2*. For example, if the term $10x_j^2$ appears in your problem, that is, if the actual coefficient of x_j^2 is *10*, then you should enter the value *20*.

You can enter data to define a QP by any of several different methods:

- ◆ You can define the problem entirely using LP format or MPS format. (*Understanding File Formats* on page 264, explains these formats in greater detail.) Briefly, the ILOG CPLEX LP format includes extensions to support quadratic objective information, and an MPS file must include a `QMATRIX` section to support quadratic data.
  - In the Interactive Optimizer, use the `read` command to read problems in from a formatted file, or use the `enter` command to enter problem data interactively.
  - For `IloCplex`, method `importModel` will seamlessly read a problem file containing QP into a Concert Technology model.
  - From the Callable Library, use the routine `CPXreadcopyprob()` to read and copy problem data from a formatted file.
- ◆ You can define the linear part of the problem by any of the entry methods described in *Put Data in the Problem Object* on page 58, and then enter the quadratic terms from an auxiliary QP file (see page 265). The QP file format is documented in greater detail in the *ILOG CPLEX Reference Manual*.

Saving QP Problems

After you enter a QP problem, whether interactively or by reading a formatted file, you can then save the problem in a formatted file. The formats available to you are LP, MPS, and SAV. When you save a QP problem in one of these formats, the quadratic information will also be recorded in the formatted file.

In addition, you can save the quadratic part of a problem in a QP file (a formatted file with the extension `.qp`, as described on page 265). To do so:

- ◆ In the Interactive Optimizer, use the `write` command
- ◆ In the Callable Library, use the routine `CPXqpwrite()`.
- ◆ Writing a QP format file is not supported by `IloCplex`.

Changing Problem Type in QPs

When you enter a problem, ILOG CPLEX determines the problem type from the available information. If you enter quadratic information about a problem, whether interactively or by reading a formatted file, then ILOG CPLEX assumes that the problem type is `qp` for quadratic.

- In the Interactive Optimizer, if you are licensed to use the ILOG CPLEX Barrier Optimizer, then you will see additional `change` options. You can use the command `change problem` with its options to change a quadratic problem to these other types:
  - `zeroed_qp` indicates that you want ILOG CPLEX to change the quadratic problem to an associated *linear relaxation* by assuming that the matrix $\boldsymbol{Q}$ is $\boldsymbol{0}$.

    When you change the problem type of a QP to `zeroed_qp`, then you can optimize the problem as an LP, using any of the ILOG CPLEX LP optimizers licensed to you (primal simplex, dual simplex, network, or barrier). This change in problem type to `zeroed_qp` retains the quadratic information about the problem, so once you have an LP solution to the relaxed LP version of the problem, you can then change the problem type back to `qp` and use the original $\boldsymbol{Q}$ matrix.

    In fact, *Diagnosing QP Infeasibility* on page 246, shows you how to use this option to diagnose infeasibilities in QPs.
  - `lp` indicates that you want ILOG CPLEX to treat the problem as an LP. This change in problem type, in contrast to `zeroed_qp`, drops the quadratic information about your problem.
  - `mip`, if you are licensed to use the MIP optimizer, indicates that you want ILOG CPLEX to treat the problem as a MIP. This change in problem type, in contrast to `zeroed_qp`, drops the quadratic information about your problem.
- From the Callable Library, use the routine `CPXchgprobtype()` to change the problem type. The header file (that is, the include file) `cplex.h` contains a section titled Problem Types of the constants that define various problem types.
- `IloCplex` handles problem types transparently (provided your license supports the required problem types). When extracting a model with a quadratic objective function, it will automatically detect it as a QP and make the required adjustments of data structures. To solve a `zeroed_qp` corresponding to an extracted QP, method `solveZeroedQP()` must be called instead of method `solve()`. With `solveZeroedQP()`, the optimizer to be used is controlled by `setRootAlgorithm()`.

Changing Quadratic Terms

ILOG CPLEX distinguishes between a *quadratic algebraic term* and a *quadratic matrix coefficient*. The quadratic algebraic terms are the coefficients that appear in the algebraic expression defined as part of the ILOG CPLEX LP format. The quadratic matrix coefficients appear in ***Q***. The quadratic coefficient of an off-diagonal term must be distributed within the ***Q*** matrix, and it is always one-half the value of the quadratic algebraic term.

To clarify that terminology, consider this example:

Minimize $a + b + 1/2(a^2 + 4ab + 7b^2)$

subject to $a + b \geq 10$

with these bounds $a \geq 0$ and $b \geq 0$

The off-diagonal quadratic algebraic term in that example is 4, so the quadratic matrix ***Q*** is

$$\begin{bmatrix} 1 & 2 \\ 2 & 7 \end{bmatrix}$$

- In a QP, you can change the quadratic matrix coefficients in the Interactive Optimizer by using the command `change qpterm`.
- From the Callable Library, use the routine `CPXchgqpcoef()` to change quadratic matrix coefficients.
- Concert Technology does not support direct editing of expressions other than linear expressions. Consequently, to change a quadratic objective function, you need to create an expression with the modified quadratic objective and use `IloObjective::setExpr()` to install this new expression in the model's objective.

Changing an off-diagonal element changes the corresponding symmetric element as well. In other words, if a call to `CPXchgqpcoef()` changes $\boldsymbol{Q}(i, j)$ to a value, it also changes $\boldsymbol{Q}(j, i)$ to that same value.

To continue our example, if we want to change the off-diagonal quadratic term from 4 to 6, we would use this sequence of commands in the Interactive Optimizer:

```
CPLEX> change qpterm
Change which quadratic term [‘variable’ ‘variable’]:
a b
Present quadratic term of variable ‘a’, variable ‘b’
is 4.000000.
Change quadratic term of variable ‘a’, variable ‘b’
to what: 6.0
Quadratic term of variable ‘a’, variable ‘b’ changed
to 6.000000.
```

From the Callable Library, the `CPXchgqpcoef()` call to change the off-diagonal term from `4` to `6` would change both of the off-diagonal matrix coefficients from `2` to `3`. Thus, the indices would be `0` and `1`, and the new matrix coefficient value would be 3.

If you have entered a linear problem without any quadratic terms, and you want to create quadratic terms, you must first *change the problem typ*e to QP. To do so, use the command `change problem qp`. This command will create an empty quadratic matrix with ***Q* = 0**.

When you change quadratic terms, there are still restrictions on the properties of the ***Q*** matrix. In a minimization problem, it must be convex, positive semi-definite. In a maximization problem, it must be concave, negative semi-definite. For example, if you change the sense of an objective function in a convex ***Q*** matrix from minimization to maximization, you will thus make the problem unsolvable. Likewise, in a convex ***Q*** matrix, if you make a term negative, you will thus make the problem unsolvable.

Optimizing QPs with the Barrier Optimizer

To use the ILOG CPLEX Barrier Optimizer in application development, you must hold a special, optional, development license. If you call barrier routines from the ILOG CPLEX Callable Library in your applications, your end user must be licensed for runtime or derived work. For more information about ILOG CPLEX licensing, contact your ILOG CPLEX representative.

To optimize a QP that you have entered or read:

◆ In the Interactive Optimizer, use the command `baropt`.

◆ From the Callable Library, use the routine `CPXbaropt()`.

◆ Method `IloCplex::solve()` will automatically invoke the barrier optimizer if the extracted model is a QP. The setting of root or node algorithm will be ignored.

For a QP, the ILOG CPLEX Barrier Optimizer generates a *pure* barrier solution. That is, the solution is *not* a basic solution. The barrier crossovers described in *Barrier Simplex Crossover* on page 131 do not apply to quadratic barrier optimizations.

The ILOG CPLEX Barrier Optimizer automatically preprocesses your quadratic problem, conducting presolution problem analysis and reductions appropriate for a QP. (It ignores the settings of the ILOG CPLEX parameters for preprocessing, presolver, and aggregator.)

Generally, the default parameter settings of the ILOG CPLEX Barrier Optimizer are appropriate for most QPs. In fact, for QPs, the ILOG CPLEX Barrier Optimizer uses only the default barrier algorithm (indicated in the Interactive Optimizer by `set barrier algorithm 0` and from the Callable Library by the parameter `CPX_PARAM_BARALG` with the value `0`). In other words, it does not use the other two algorithms discussed in the context of *linear* barrier optimization and listed in Table 4.13 on page 145.

Understanding QP Solution Information

When the ILOG CPLEX Barrier Optimizer reaches a solution for a QP, it generates all the primal and dual information available for pure, nonbasis barrier solutions, as described in *Understanding Solution Quality from the Barrier LP Optimizer* on page 138.

Reminder: Since there is no basic solution after this kind of optimization, there is no objective range information. Also, since there is no basic solution, it is not possible to save an advanced basis for restarts. You cannot write `.bin` nor `.txt` files either.

You can save QP solution information as VEC files with the extension `.vec`.

◆ In the Interactive Optimizer, use the `write` command followed by a file name with the extension `.vec`.

◆ From the Callable Library, use the routine `CPXvecwrite()`.

To display information about a QP solution from the ILOG CPLEX Barrier Optimizer:

◆ In the Interactive Optimizer, there are several options:

  - `display solution quality` provides information about the quality of a QP solution with respect to solution optimality. (Table 4.11 on page 138 lists and explains this information.)
  - `display problem variable` shows the quadratic objective function coefficient of a specific variable.
  - `display problem qpvariables` shows the names of quadratic variables.
  - `display problem constraint obj` returns the complete linear and quadratic objective function.

◆ From the Callable Library, use the routine `CPXsolution()` to access the solution values and the routine `CPXgetdblquality()` to access information about the quality of the solution.

◆ From an `IloCplex` object, solution information can be queried as for any other problem type using the `getValues()` and similar methods. Also, method `getQuality()` works the same way for QPs as for any other problem type.

Tuning QP Performance

As we mentioned, the default settings of the parameters controlling the ILOG CPLEX Barrier Optimizer are appropriate for most QPs. However, if you need to experiment with those settings to tune performance for your particular problem, we recommend that you review *Tuning Barrier Optimizer Performance* on page 140, where we explain those parameters in the context of *linear* optimization.

Diagnosing QP Infeasibility

If the ILOG CPLEX Barrier Optimizer reports that your QP is *primal infeasible*, then you can ask ILOG CPLEX to *relax* the QP to its *linear* version with the ***Q*** matrix set to **0**.

To change the problem type:

- In the Interactive Optimizer, use the command `change problem zeroed_qp`.
- From the Callable Library, use the routine `CPXchgprobtype()`

You can then solve the relaxed linear version by means of a ILOG CPLEX simplex optimizer, such as primal simplex or dual simplex. Then you can apply the ILOG CPLEX infeasibility finder to that relaxed solution, with its associated, original QP information, to help you diagnose the source of the infeasibility. (*Diagnosing LP Infeasibility* on page 112 explains how to use the ILOG CPLEX infeasibility finder following a simplex optimizer.)

Since `IloCplex` handles problem types transparently, the way to diagnose an infeasible model is slightly different. Since infeasibility does not depend on the objective function, you start by removing the objective extractable from the extracted model. This way, the model seen by the `cplex` object is an LP with a 0 objective and the LP IIS finder can be applied. To get the original model back, simply add the objective back to the model.

Example: Creating a QP, Optimizing, Finding a Solution

This example shows you how to build and solve a QP. The problem being created and solved is:

Maximize

$$x_1 + 2x_2 + 3x_3 - 0.5\ (33x_1^2 + 22x_2^2 + 11x_3^2 - 12x_1x_2 - 23x_2x_3)$$

subject to

$$-x_1 + x_2 + x_3 \leq 20$$
$$x_1 - 3x_2 + x_3 \leq 30$$

with these bounds

$$0 \leq x_1 \leq 40$$
$$0 \leq x_2 \leq +\infty$$
$$0 \leq x_3 \leq +\infty$$

Example: iloqpex1.cpp

This example is almost identical to `ilolpex1.cpp` with only function `populatebyrow` to create the model. Also, this function differs only in the creation of the objective from its `ilolpex1.cpp` counterpart. Here the objective function is created and added to the model like this:

```
   model.add(IloMaximize(env, x[0] + 2 * x[1] + 3 * x[2]
                 - 0.5 * (33*x[0]*x[0] + 22*x[1]*x[1] + 11*x[2]*x[2]
                               - 12*x[0]*x[1] - 23*x[1]*x[2]) ));
```

In general, any expression built of basic operations `+`, `-`, `*`, `/` constant, and brackets '`()`' that amounts to a quadratic and optional linear term can be used for building QP objective function. Note that the expressions of the objective or any constraint of the model must not contain `IloPiecewiselinear` when a QP objective is specified, in order for `IloCplex` to be able to process the model.

Complete Program: iloqpex1.cpp

The complete program, `iloqpex1.cpp`, appears here or online in the standard distribution.

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

static void
   populatebyrow     (IloModel model, IloNumVarArray var, IloRangeArray con);

int
main (int argc, char **argv)
{
   IloEnv   env;
   try {
      IloModel model(env);
      IloNumVarArray var(env);
      IloRangeArray con(env);

      populatebyrow (model, var, con);

      IloCplex cplex(model);

      // Optimize the problem and obtain solution.
      if ( !cplex.solve() ) {
         env.error() << “Failed to optimize LP” << endl;
         throw(-1);
      }

      IloNumArray vals(env);
      env.out() << “Solution status = “ << cplex.getStatus() << endl;
      env.out() << “Solution value  = “ << cplex.getObjValue() << endl;
      cplex.getValues(vals, var);
      env.out() << “Values        = “ << vals << endl;
      cplex.getSlacks(vals, con);
      env.out() << “Slacks        = “ << vals << endl;
      cplex.getDuals(vals, con);
      env.out() << “Duals         = “ << vals << endl;
      cplex.getReducedCosts(vals, var);
      env.out() << “Reduced Costs = “ << vals << endl;

      cplex.exportModel(“qpex1.lp”);
   }
   catch (IloException& e) {
      cerr << “Concert exception caught: “ << e << endl;
   }
   catch (...) {
      cerr << “Unknown exception caught” << endl;
   }

   env.end();

   return 0;
}  // END main


// To populate by row, we first create the variables, and then use them to
// create the range constraints and objective.  The model we create is:
//
//    Maximize
//     obj: x1 + 2 x2 + 3 x3
//            - 0.5 ( 33*x1*x1 + 22*x2*x2 + 11*x3*x3
//                             - 12*x1*x2 - 23*x2*x3 )
//    Subject To
//     c1: - x1 + x2 + x3 <= 20
//     c2: x1 - 3 x2 + x3 <= 30
//    Bounds
//     0 <= x1 <= 40
//    End

static void
populatebyrow (IloModel model, IloNumVarArray x, IloRangeArray c)
{
   IloEnv env = model.getEnv();

   x.add(IloNumVar(env, 0.0, 40.0));
   x.add(IloNumVar(env));
   x.add(IloNumVar(env));
   model.add(IloMaximize(env, x[0] + 2 * x[1] + 3 * x[2]
                            - 0.5 * (33*x[0]*x[0] + 22*x[1]*x[1] + 11*x[2]*x[2]
                                                  - 12*x[0]*x[1] -
23*x[1]*x[2]) ));

   c.add( - x[0] +     x[1] + x[2] <= 20);
   c.add(   x[0] - 3 * x[1] + x[2] <= 30);
   model.add(c);

}  // END populatebyrow
```

Example: qpex1.c

In the routine `setproblemdata()`, there are parameters for `qmatbeg`, `qmatcnt`, `qmatind`, and `qmatval` to fill the quadratic coefficient matrix. The Callable Library routine `CPXcopyquad()` copies this data into the problem object created by the Callable Library routine `CPXcreateprob()`.

In this example, the problem is a *maximization*, so we handle that fact by specifying the objective sense of `CPX_MAX`.

The off-diagonal terms in the matrix ***Q*** are one-half the value of the terms x_1x_2, and x_2x_3 as they appear in the algebraic form of the example.

Instead of calling `CPXlpopt()` to find a solution as we do for the *linear* programming problem in `lpex1.c`, this time we call `CPXbaropt()` to optimize this *quadratic* programming problem.

Complete Program: qpex1.c

The complete program, `qpex1.c`, appears here or online in the standard distribution.

```
#include <ilcplex/cplex.h>
#include <stdlib.h>

/* Bring in the declarations for the string functions */

#include <string.h>

/* Include declaration for function at end of program */

#ifndef  CPX_PROTOTYPE_MIN

static int
   setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                   int *objsen_p, double **obj_p, double **rhs_p,
                   char **sense_p, int **matbeg_p, int **matcnt_p,
                   int **matind_p, double **matval_p, double **lb_p,
                   double **ub_p, int **qmatbeg_p, int **qmatcnt_p,
                   int **qmatind_p, double **qmatval_p);

static void
   free_and_null (char **ptr);
#else

static int
   setproblemdata ();

static void
   free_and_null ();

#endif


/* The problem we are optimizing will have 2 rows, 3 columns,
   6 nonzeros, and 7 nonzeros in the quadratic coefficient matrix. */

#define NUMROWS    2
#define NUMCOLS    3
#define NUMNZ      6
#define NUMQNZ     7

#ifndef  CPX_PROTOTYPE_MIN
int
main (void)
#else
int
main ()
```

```
#endif
{
   /* Declare pointers for the variables and arrays that will contain
      the data which define the LP problem.  The setproblemdata() routine
      allocates space for the problem data.  */

   char     *probname = NULL;
   int      numcols;
   int      numrows;
   int      objsen;
   double   *obj = NULL;
   double   *rhs = NULL;
   char     *sense = NULL;
   int      *matbeg = NULL;
   int      *matcnt = NULL;
   int      *matind = NULL;
   double   *matval = NULL;
   double   *lb = NULL;
   double   *ub = NULL;
   int      *qmatbeg = NULL;
   int      *qmatcnt = NULL;
   int      *qmatind = NULL;
   double   *qmatval = NULL;

   /* Declare and allocate space for the variables and arrays where we
      will store the optimization results including the status, objective
      value, variable values, dual values, row slacks and variable
      reduced costs. */

   int      solstat;
   double   objval;
   double   x[NUMCOLS];
   double   pi[NUMROWS];
   double   slack[NUMROWS];
   double   dj[NUMCOLS];


   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status;
   int           i, j;
   int           cur_numrows, cur_numcols;

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no output,
      so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
      be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

   if ( env == NULL ) {
   char  errmsg[1024];
      fprintf (stderr, “Could not open CPLEX environment.\n”);
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, “%s”, errmsg);
      goto TERMINATE;
   }

   /* Turn on output to the screen */

   status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
   if ( status ) {
      fprintf (stderr,
               “Failure to turn on screen indicator, error %d.\n”, status);
      goto TERMINATE;
   }

   /* Fill in the data for the problem.  */

   status = setproblemdata (&probname, &numcols, &numrows, &objsen, &obj,
                            &rhs, &sense, &matbeg, &matcnt, &matind,
                            &matval, &lb, &ub, &qmatbeg, &qmatcnt,
                            &qmatind, &qmatval);
   if ( status ) {
      fprintf (stderr, “Failed to build problem data arrays.\n”);
      goto TERMINATE;
   }

   /* Create the problem. */

   lp = CPXcreateprob (env, &status, probname);

   /* A returned pointer of NULL may mean that not enough memory
      was available or there was some other problem.  In the case of
      failure, an error message will have been written to the error
      channel from inside CPLEX.  In this example, the setting of
      the parameter CPX_PARAM_SCRIND causes the error message to
      appear on stdout.  */

   if ( lp == NULL ) {
      fprintf (stderr, “Failed to create problem.\n”);
      goto TERMINATE;
   }

   /* Now copy the LP part of the problem data into the lp */

   status = CPXcopylp (env, lp, numcols, numrows, objsen, obj, rhs,
                       sense, matbeg, matcnt, matind, matval,
                       lb, ub, NULL);

   if ( status ) {
      fprintf (stderr, “Failed to copy problem data.\n”);
      goto TERMINATE;
   }

   status = CPXcopyquad (env, lp, qmatbeg, qmatcnt, qmatind, qmatval);
   if ( status ) {
      fprintf (stderr, “Failed to copy quadratic matrix.\n”);
      goto TERMINATE;
   }

   /* Optimize the problem and obtain solution. */
```

```
   status = CPXbaropt (env, lp);
   if ( status ) {
      fprintf (stderr, “Failed to optimize QP.\n”);
      goto TERMINATE;
   }

   status = CPXsolution (env, lp, &solstat, &objval, x, pi, slack, dj);
   if ( status ) {
      fprintf (stderr, “Failed to obtain solution.\n”);
      goto TERMINATE;
   }


   /* Write the output to the screen. */

   printf (“\nSolution status = %d\n”, solstat);
   printf (“Solution value  = %f\n\n”, objval);

   /* The size of the problem should be obtained by asking CPLEX what
      the actual size is, rather than using what was passed to CPXcopylp.
      cur_numrows and cur_numcols store the current number of rows and
      columns, respectively.  */

   cur_numrows = CPXgetnumrows (env, lp);
   cur_numcols = CPXgetnumcols (env, lp);
   for (i = 0; i < cur_numrows; i++) {
      printf (“Row %d:  Slack = %10f  Pi = %10f\n”, i, slack[i], pi[i]);
   }

   for (j = 0; j < cur_numcols; j++) {
      printf (“Column %d:  Value = %10f  Reduced cost = %10f\n”,
              j, x[j], dj[j]);
   }

   /* Finally, write a copy of the problem to a file. */

   status = CPXwriteprob (env, lp, “qpex1.lp”, NULL);
   if ( status ) {
      fprintf (stderr, “Failed to write LP to disk.\n”);
      goto TERMINATE;
   }

TERMINATE:

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, “CPXfreeprob failed, error code %d.\n”, status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);
      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, “Could not close CPLEX environment.\n”);
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, “%s”, errmsg);
      }
   }

   /* Free up the problem data arrays, if necessary. */

   free_and_null ((char **) &probname);
   free_and_null ((char **) &obj);
   free_and_null ((char **) &rhs);
   free_and_null ((char **) &sense);
   free_and_null ((char **) &matbeg);
   free_and_null ((char **) &matcnt);
   free_and_null ((char **) &matind);
   free_and_null ((char **) &matval);
   free_and_null ((char **) &lb);
   free_and_null ((char **) &ub);
   free_and_null ((char **) &qmatbeg);
   free_and_null ((char **) &qmatcnt);
   free_and_null ((char **) &qmatind);
   free_and_null ((char **) &qmatval);

   return (status);

}  /* END main */


/* This function fills in the data structures for the quadratic program:

      Maximize
       obj: x1 + 2 x2 + 3 x3
              - 0.5 ( 33x1*x1 + 22*x2*x2 + 11*x3*x3
                   -  12*x1*x2 - 23*x2*x3 )
      Subject To
       c1: - x1 + x2 + x3 <= 20
       c2: x1 - 3 x2 + x3 <= 30
      Bounds
       0 <= x1 <= 40
      End
 */


#ifndef  CPX_PROTOTYPE_MIN
static int
setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                int *objsen_p, double **obj_p, double **rhs_p,
                char **sense_p, int **matbeg_p, int **matcnt_p,
                int **matind_p, double **matval_p, double **lb_p,
                double **ub_p, int **qmatbeg_p, int **qmatcnt_p,
                int **qmatind_p, double **qmatval_p)
#else
static int
setproblemdata (probname_p, numcols_p, numrows_p, objsen_p, obj_p,
                rhs_p, sense_p, matbeg_p, matcnt_p, matind_p, matval_p,
                lb_p, ub_p, qmatbeg_p, qmatcnt_p, qmatind_p, qmatval_p)
char    **probname_p;
int     *numcols_p;
int     *numrows_p;
int     *objsen_p;
double  **obj_p;
double  **rhs_p;
char    **sense_p;
int     **matbeg_p;
int     **matcnt_p;
int     **matind_p;
double  **matval_p;
double  **lb_p;
double  **ub_p;
int     **qmatbeg_p;
int     **qmatcnt_p;
int     **qmatind_p;
double  **qmatval_p;
#endif
{
   char     *zprobname = NULL;     /* Problem name <= 16 characters */
   double   *zobj = NULL;
   double   *zrhs = NULL;
   char     *zsense = NULL;
   int      *zmatbeg = NULL;
   int      *zmatcnt = NULL;
   int      *zmatind = NULL;
   double   *zmatval = NULL;
   double   *zlb = NULL;
   double   *zub = NULL;
   int      *zqmatbeg = NULL;
   int      *zqmatcnt = NULL;
   int      *zqmatind = NULL;
   double   *zqmatval = NULL;
   int      status = 0;

   zprobname = (char *) malloc (16 * sizeof(char));
   zobj      = (double *) malloc (NUMCOLS * sizeof(double));
   zrhs      = (double *) malloc (NUMROWS * sizeof(double));
   zsense    = (char *) malloc (NUMROWS * sizeof(char));
   zmatbeg   = (int *) malloc (NUMCOLS * sizeof(int));
   zmatcnt   = (int *) malloc (NUMCOLS * sizeof(int));
   zmatind   = (int *) malloc (NUMNZ * sizeof(int));
   zmatval   = (double *) malloc (NUMNZ * sizeof(double));
   zlb       = (double *) malloc (NUMCOLS * sizeof(double));
   zub       = (double *) malloc (NUMCOLS * sizeof(double));
   zqmatbeg  = (int *) malloc (NUMCOLS * sizeof(int));
   zqmatcnt  = (int *) malloc (NUMCOLS * sizeof(int));
   zqmatind  = (int *) malloc (NUMQNZ * sizeof(int));
   zqmatval  = (double *) malloc (NUMQNZ * sizeof(double));

   if ( zprobname == NULL || zobj     == NULL ||
        zrhs      == NULL || zsense   == NULL ||
        zmatbeg   == NULL || zmatcnt  == NULL ||
        zmatind   == NULL || zmatval  == NULL ||
        zlb       == NULL || zub      == NULL ||
        zqmatbeg  == NULL || zqmatcnt == NULL ||
        zqmatind  == NULL || zqmatval == NULL  )  {
      status = 1;
      goto TERMINATE;
   }

   strcpy (zprobname, “example”);

   /* The code is formatted to make a visual correspondence
      between the mathematical linear program and the specific data
      items.   */

     zobj[0]  = 1.0;   zobj[1]   = 2.0;   zobj[2] = 3.0;

   zmatbeg[0] = 0;     zmatbeg[1] = 2;    zmatbeg[2] = 4;
   zmatcnt[0] = 2;     zmatcnt[1] = 2;    zmatcnt[2] = 2;

   zmatind[0] = 0;     zmatind[2] = 0;    zmatind[4] = 0;     zsense[0] = ‘L’;
   zmatval[0] = -1.0;  zmatval[2] = 1.0;  zmatval[4] = 1.0;   zrhs[0]   = 20.0;

   zmatind[1] = 1;     zmatind[3] = 1;    zmatind[5] = 1;     zsense[1] = ‘L’;
   zmatval[1] = 1.0;   zmatval[3] = -3.0; zmatval[5] = 1.0;   zrhs[1]   = 30.0;

       zlb[0] = 0.0;       zlb[1] = 0.0;          zlb[2] = 0.0;
       zub[0] = 40.0;      zub[1] = CPX_INFBOUND; zub[2] = CPX_INFBOUND;

   /* Now set up the Q matrix.  Note that we set the values knowing that
    * we’re doing a maximization problem, so negative values go on
    * the diagonal.  Also, the off diagonal terms are each repeated,
    * by taking the algebraic term and dividing by 2 */

   zqmatbeg[0] = 0;     zqmatbeg[1] = 2;     zqmatbeg[2] = 5;
   zqmatcnt[0] = 2;     zqmatcnt[1] = 3;     zqmatcnt[2] = 2;

   /* Matrix is set up visually.  Note that the x1*x3 term is 0, and is
    * left out of the matrix.  */

   zqmatind[0] = 0;     zqmatind[2] = 0;
   zqmatval[0] = -33.0; zqmatval[2] = 6.0;

   zqmatind[1] = 1;     zqmatind[3] = 1;     zqmatind[5] = 1;
   zqmatval[1] = 6.0;   zqmatval[3] = -22.0; zqmatval[5] = 11.5;

                        zqmatind[4] = 2;     zqmatind[6] = 2;
                        zqmatval[4] = 11.5;  zqmatval[6] = -11.0;

TERMINATE:

   if ( status ) {
      free_and_null ((char **) &zprobname);
      free_and_null ((char **) &zobj);
      free_and_null ((char **) &zrhs);
      free_and_null ((char **) &zsense);
      free_and_null ((char **) &zmatbeg);
      free_and_null ((char **) &zmatcnt);
      free_and_null ((char **) &zmatind);
      free_and_null ((char **) &zmatval);
      free_and_null ((char **) &zlb);
      free_and_null ((char **) &zub);
      free_and_null ((char **) &zqmatbeg);
      free_and_null ((char **) &zqmatcnt);
      free_and_null ((char **) &zqmatind);
      free_and_null ((char **) &zqmatval);
   }
   else {
      *numcols_p   = NUMCOLS;
      *numrows_p   = NUMROWS;
      *objsen_p    = CPX_MAX;   /* The problem is maximization */

      *probname_p  = zprobname;
      *obj_p       = zobj;
      *rhs_p       = zrhs;
      *sense_p     = zsense;
      *matbeg_p    = zmatbeg;
      *matcnt_p    = zmatcnt;
      *matind_p    = zmatind;
      *matval_p    = zmatval;
      *lb_p        = zlb;
      *ub_p        = zub;
      *qmatbeg_p   = zqmatbeg;
      *qmatcnt_p   = zqmatcnt;
      *qmatind_p   = zqmatind;
      *qmatval_p   = zqmatval;
   }
   return (status);

}  /* END setproblemdata */



/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

#ifndef  CPX_PROTOTYPE_MIN
static void
free_and_null (char **ptr)
#else
static void
free_and_null (ptr)
char  **ptr;
#endif
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */
```

Example: Reading a QP from a File

This example shows you how to optimize a QP with routines from the ILOG CPLEX Callable Library when the problem data is stored in a file. The example derives from `lpex2.c`, described in the manual *ILOG CPLEX Getting Started*.

This example differs from `lpex2.c` in its command line. In `qpex2.c`, there is no need of a command-line argument to indicate which optimizer to call, as only the ILOG CPLEX Barrier Optimizer is used to solve QPs. In other words, this example always calls the routine `CPXbaropt()`.

This example also differs in the way it shows a solution. Since no basis is available for the QP, this example calls the routine `CPXgetx()` to get a solution. It is, however, possible to call `CPXsolution()` to get a primal and dual solution to the problem.

Like other applications based on the ILOG CPLEX Callable Library, this one begins with calls to `CPXopenCPLEX()` to initialize the ILOG CPLEX environment and to `CPXcreateprob()` to create the problem object. Before it ends, it frees the problem object with a call to `CPXfreeprob()`, and it frees the environment with a call to `CPXcloseCPLEX()`.

Complete Program: qpex2.c

The complete program, `qpex2.c`, appears here or online in the standard distribution.

```
#include <ilcplex/cplex.h>

/* Bring in the declarations for the string and character functions
   and malloc */

#include <ctype.h>
#include <stdlib.h>
#include <string.h>

/* Include declarations for functions in this program */

#ifndef  CPX_PROTOTYPE_MIN

static void
   free_and_null (char **ptr),
   usage         (char *progname);

#else

static void
   free_and_null (),
   usage         ();

#endif

#ifndef  CPX_PROTOTYPE_MIN
```

```
int
main (int argc, char *argv[])
#else
int
main (argc, argv)
int   argc;
char  *argv[];
#endif
{
   /* Declare and allocate space for the variables and arrays where we
      will store the optimization results including the status, objective
      value and variable values. */

   int      solstat;
   double   objval;
   double   *x = NULL;

   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status;
   int           j;
   int           cur_numcols;

   /* Check the command line arguments */

   if ( argc != 2 ) {
      usage (argv[0]);
      goto TERMINATE;
   }

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no output,
      so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
      be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

   if ( env == NULL ) {
   char  errmsg[1024];
      fprintf (stderr, “Could not open CPLEX environment.\n”);
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, “%s”, errmsg);
      goto TERMINATE;
   }

   /* Turn on output to the screen */

   status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
   if ( status != 0 ) {
      fprintf (stderr,
               “Failure to turn on screen indicator, error %d.\n”, status);
      goto TERMINATE;
   }
```

```
/* Create the problem, using the filename as the problem name */

lp = CPXcreateprob (env, &status, argv[1]);

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.  Note that most CPLEX routines return
   an error code to indicate the reason for failure.   */

if ( lp == NULL ) {
   fprintf (stderr, “Failed to create LP.\n”);
   goto TERMINATE;
}

/* Now read the file, and copy the data into the created lp */

status = CPXreadcopyprob (env, lp, argv[1], NULL);
if ( status ) {
   fprintf (stderr, “Failed to read and copy the problem data.\n”);
   goto TERMINATE;
}

if ( CPXgetprobtype (env, lp) != CPXPROB_QP ) {
   fprintf (stderr, “File does not contain quadratic data.  Exiting.\n”);
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */

status = CPXbaropt (env, lp);

if ( status ) {
   fprintf (stderr, “Failed to optimize QP.\n”);
   goto TERMINATE;
}

solstat = CPXgetstat (env, lp);
printf (“Solution status %d.\n”, solstat);

status  = CPXgetobjval (env, lp, &objval);

if ( status ) {
   fprintf (stderr,”Failed to obtain objective value.\n”);
   goto TERMINATE;
}

printf (“Objective value %.10g\n”, objval);

/* The size of the problem should be obtained by asking CPLEX what
   the actual size is.  cur_numcols stores the current number
   of columns. */

cur_numcols = CPXgetnumcols (env, lp);

/* Allocate space for solution */
```

```
   x     = (double *) malloc (cur_numcols*sizeof(double));

   if ( x == NULL ) {
      fprintf (stderr,”No memory for basis statuses.\n”);
      goto TERMINATE;
   }

   status = CPXgetx (env, lp, x, 0, cur_numcols-1);
   if ( status ) {
      fprintf (stderr, “Failed to obtain solution.\n”);
      goto TERMINATE;
   }

   /* Write out the solution */

   for (j = 0; j < cur_numcols; j++) {
      printf ( “Column %d:  Value = %17.10g\n”, j, x[j]);
   }


TERMINATE:

   /* Free up the basis and solution */

   free_and_null ((char **) &x);

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, “CPXfreeprob failed, error code %d.\n”, status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
      char  errmsg[1024];
         fprintf (stderr, “Could not close CPLEX environment.\n”);
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, “%s”, errmsg);
      }
   }

   return (status);

}  /* END main */
```

```
/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

#ifndef  CPX_PROTOTYPE_MIN
static void
free_and_null (char **ptr)
#else
static void
free_and_null (ptr)
char  **ptr;
#endif
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */


#ifndef  CPX_PROTOTYPE_MIN
static void
usage (char *progname)
#else
static void
usage (progname)
char *progname;
#endif
{
   fprintf (stderr,"Usage: %s filename\n", progname);
   fprintf (stderr,"   where filename is a file with extension \n");
   fprintf (stderr,"      MPS, SAV, or LP (lower case is allowed)\n");
   fprintf (stderr,"  This program uses the CPLEX Barrier optimizer\n");
   fprintf (stderr,"    to optimize quadratic programs.\n");
   fprintf (stderr," Exiting...\n");
} /* END usage */
```

C H A P T E R

8

More About Using ILOG CPLEX

This chapter provides information designed to help you master several important aspects of ILOG CPLEX. It includes sections on:

- Managing Input & Output
- Using Query Routines
- Using Callbacks
- Using Parallel Optimizers

Managing Input & Output

This section tells you about input to and output from ILOG CPLEX. It contains the following subsections:

- Understanding File Formats
- Managing Log Files: the Log File Parameter
- Handling Message Channels: the Output-Channel Parameter
- Handling Message Channels: Callable Library Routines
- Example: Using the Message Handler

Understanding File Formats

The *ILOG CPLEX Reference Manual* documents the file formats that ILOG CPLEX supports. Here is a brief description of these file formats:

- ◆ BAS files are text files governed by MPS conventions (that is, they are not binary) for saving a problem basis.
- ◆ BIN files are binary files. ILOG CPLEX uses this format when it writes solution files containing the binary representation of real numbers.
- ◆ DPE is the format ILOG CPLEX uses to write a problem in a binary SAV file after the objective of a problem has been perturbed for use with the dual simplex optimizer.
- ◆ DUA format, governed by MPS conventions, writes the dual formulation of a problem currently in memory so that the MPS file can later be read back in and the dual formulation can then be optimized explicitly. This file format is largely obsolete now since you can use the command `set preprocessing dual` in the Interactive Optimizer to tell ILOG CPLEX to solve the dual formulation of an LP automatically. (You no longer have to tell ILOG CPLEX to write the dual formulation to a DUA file and then tell ILOG CPLEX to read the file back in and solve it.)
- ◆ EMB is the format ILOG CPLEX uses to save an embedded network it extracts from a problem. EMB files are written in MPS format.
- ◆ IIS is the format ILOG CPLEX uses to represent irreducible inconsistent sets of constraints. *Finding a Set of Irreducibly Inconsistent Constraints* on page 116 and *Example: Writing an IIS-Type File* on page 118 explain more about these kinds of files.
- ◆ LP (Linear Programming) is a ILOG CPLEX-specific file formatted for entering problems in an algebraic, row-oriented form. In other words, LP format allows you to enter problems in terms of their constraints. When you enter problems interactively in the Interactive Optimizer, you are implicitly using LP format. ILOG CPLEX will also read in files in LP format. *Working with LP Files* on page 266 explains more fully how to use LP files with ILOG CPLEX.
- ◆ MIN format for representing minimum-cost network-flow problems was introduced by DIMACS in 1991. More information about DIMACS network file formats is available via anonymous ftp from `ftp://dimacs.rutgers.edu/pub/netflow/general-info/specs.tex`.
- ◆ MPS (Mathematical Programming System) is an industry-standard, ASCII-text file format for mathematical programming problems. Besides the industry conventions, ILOG CPLEX also supports extensions to this format for ILOG CPLEX-specific cases, such as names of more than eight characters, blank space as delimiters between columns, etc. *Working with MPS Files* on page 267 in this manual explains more fully how to use MPS files with ILOG CPLEX.
- ◆ MST is a text format ILOG CPLEX uses to enter a starting solution for a MIP.

- NET is a ILOG CPLEX-specific ASCII format for network-flow problems. It is flexible and supports named nodes and arcs.
- ORD is a format available only if you are licensed to use the ILOG CPLEX MIP Optimizer. It is used to enter and to save priority orders for branching. It may contain branching instructions for individual variables.
- PPE is the format ILOG CPLEX uses to write a problem in a binary SAV file after the bounds of a problem have been perturbed for use with the primal simplex optimizer.
- PRE is the format ILOG CPLEX uses to write a presolved, reduced problem formulation to a binary SAV file. Since a presolved problem has been reduced, it will not correspond to the original problem.
- QP is a format available only if you are licensed to use the ILOG CPLEX Barrier Optimizer. It contains the coefficients of nonzero coefficients in the ***Q*** matrix of a quadratic programming problem. You must enter the *linear* part of that quadratic programming problem *first* (by whichever means you choose).
- REW is a format to write a problem in MPS format with disguised row and column names. This format may be useful, for example, for problems that you consider highly proprietary.
- SAV is a ILOG CPLEX-specific *binary* format for reading and writing problems and their associated basis information. ILOG CPLEX includes the basis in a SAV file only if the problem currently in memory has been optimized and a basis exists. This format offers the advantage of being numerically accurate (to the same degree as your platform) in contrast to text file formats that may lose numerical accuracy. It also has the additional benefit of being efficient with respect to read and write time. However, since a SAV file is binary, you cannot read nor edit it with your favorite text editor.
- SOS is a format available only if you are licensed to use the ILOG CPLEX MIP Optimizer. It declares special ordered sets, the set branching order, and weights for each set member.
- TRE is a format available only if you are licensed to use the ILOG CPLEX MIP Optimizer. It saves information about progress through the branch & cut tree. It is a binary format.
- TXT files are ASCII-text files. ILOG CPLEX uses this format when it writes solution files in text.
- VEC is a format available only if you are licensed to use the ILOG CPLEX Barrier Optimizer. It saves the solution to a pure barrier optimization prior to crossover (that is, a nonbasis solution) that can later be read back in and used to initiate crossover. *Using VEC File Format* on page 134 explains how to use this file format.

Working with LP Files

LP files are row-oriented so you can look at a problem as you enter it in a naturally and intuitively algebraic way. However, ILOG CPLEX represents a problem internally in a column-ordered format. This difference between the way ILOG CPLEX accepts a problem in LP format and the way it stores the problem internally may have an impact on *memory use* and on the *order* in which *variables* are displayed on screen or in files.

Memory Use and LP Files

Whenever ILOG CPLEX reads a file in LP format, it converts from row-orientation to column-orientation. The conversion requires memory. On a platform with limited memory, if there is insufficient memory to read a given problem in LP format, there may still be sufficient memory to read the problem in MPS format. Generally, the ILOG CPLEX MPS file reader will load MPS format files more efficiently than will the LP reader loading LP format files.

Variable Order and LP Files

As ILOG CPLEX reads an LP format file by rows, it adds columns as it encounters them in a row. This convention will have an impact on the order in which variables are named and displayed. For example, consider this problem:

$$
\begin{array}{llllllll}
\text{Maximize} & & & 2x_2 & + & 3x_3 & & \\
\text{subject to} & & & & & & & \\
 & -x_1 & + & x_2 & + & x_3 & \leq & 20 \\
 & x_1 & - & 3x_2 & + & x_3 & \leq & 30 \\
\text{with these bounds} & & & & & & & \\
 & 0 & \leq & x_1 & \leq & 40 & & \\
 & 0 & \leq & x_2 & \leq & +\infty & & \\
 & 0 & \leq & x_3 & \leq & +\infty & &
\end{array}
$$

Since ILOG CPLEX reads the objective function as the first row, the two columns appearing there will become the first two variables. When the problem is displayed or rewritten into another LP file, the variables there will appear in a different order within each row. In this example, if you execute the command `display problem all`, you will see this:

```
Maximize
 obj: 2 x2 + 3 x3
Subject To
 c1: x2 + x3 - x1 <= 20
 c2: - 3 x2 + x3 + x1 <= 30
Bounds
 0 <= x1 <= 40
 All other variables are >= 0.
```

That is, `x1` appears at the end of each constraint in which it has a nonzero coefficient. This re-ordering does not affect the optimal value of the problem, but you may find it disconcerting when you first encounter it.

Working with MPS Files

The ILOG CPLEX MPS file reader is highly compatible with existing modeling systems. There is generally no need to modify existing problem files to use them with ILOG CPLEX. However, there are ILOG CPLEX-specific conventions that may be useful for you to know. This section explains those conventions, and the *ILOG CPLEX Reference Manual* documents MPS format more fully.

Free Rows in MPS Files

In an MPS file, ILOG CPLEX selects the first free row or N-type row as the objective function, and it discards all subsequent free rows unless it is instructed otherwise by an `OBJNAME` section in the file. To retain free rows in an MPS file, reformulate them as equality rows with an additional free variable. For example, replace the free row `x + y` by the equality row `x + y - s = 0` where `s` is free. Generally, the ILOG CPLEX presolver will remove rows like that before optimization so they will have no impact on performance.

Ranged Rows in MPS Files

To handle ranged rows, ILOG CPLEX introduces a temporary range variable, creates appropriate bounds for this variable, and changes the sense of the row to an equality (that is, MPS type EQ). The added range variables will have the same name as the ranged row with the characters `Rg` prefixed. When ILOG CPLEX generates solution reports, it removes these temporary range variables from the constraint matrix.

Extra Rim Vectors in MPS Files

The MPS format allows multiple right-hand sides (RHS), multiple bounds, and multiple range vectors. It also allows extra free rows. Together, these features are known as *extra rim vectors*. By default, the ILOG CPLEX MPS reader selects the first RHS, bound, and range definitions that it finds. The first free row (that is, N-type row) becomes the objective function, and the remaining free rows are discarded. The extra rim data are also discarded.

Naming Conventions in MPS Files

ILOG CPLEX accepts any noncontrol-character within a name. However, ILOG CPLEX recognizes blanks (that is, space) as delimiters, so you must avoid them in names. You should also avoid `$` (dollar sign) and `*` (asterisk) as characters in names because they normally indicate a comment within a data record.

Error Checking in MPS Files

Fairly common problems in MPS files include split vectors, unnamed columns, and duplicated names. ILOG CPLEX checks for these conditions and reports them. If repeated rows or columns occur in an MPS file, ILOG CPLEX reports an error and stops reading the file. You can then edit the MPS file to correct the source of the problem.

Saving Modified MPS Files

You may often want to save a modified MPS file for later use. To that end, ILOG CPLEX will write out a problem exactly as it appears in memory. All your revisions of that problem will appear in the new file. One potential area for confusion occurs when a maximization problem is saved. Since MPS conventionally represents all problems as minimizations, ILOG CPLEX reverses the sign of the objective-function coefficients when it writes a maximization problem to an MPS file. When you read and optimize this new problem, the values of the variables will be valid for the original model. However, since the problem has been converted from a maximization to the equivalent minimization, the objective, dual, and reduced-cost values will have reversed signs.

Converting File Formats

MPS, Mathematical Programming System, an industry-standard format based on ASCII-text has historically been restricted to a fixed format in which data fields were limited to eight characters and specific fields had to appear in specific columns on specific lines. ILOG CPLEX supports extensions to MPS that allow more descriptive names (that is, more than eight characters), greater accuracy for numerical data, and greater flexibility in data positions.

Most MPS files in fixed format conform to the ILOG CPLEX extensions and thus can be read by the ILOG CPLEX MPS reader without error. However, the ILOG CPLEX MPS reader will *not* accept the following conventions:

- blank space within a name;
- blank lines;
- missing fields (such as bound names and right-hand side names);
- extraneous, uncommented characters;
- blanks in lieu of repeated name fields, such as bound vector names and right-hand side names.

You can convert fixed-format MPS files that contain those conventions into acceptable ILOG CPLEX-extended MPS files. To do so, use the `convert` utility supplied in the standard distribution of ILOG CPLEX. The `convert` utility removes unreadable features from fixed-format MPS, REW, BAS, SOS, and ORD files. It runs from the operating system prompt of your platform. Here is the syntax of the convert utility:

```
convert -option inputfilename outputfilename
```

Your command must include an input-file name and an output-file name; they must be different from each other. The options, summarized in Table 8.1, indicate the file type. You may specify only one option. If you do not specify an option, ILOG CPLEX attempts to deduce the file type from the extension in the file name.

Table 8.1 *Options for the* `convert` *Utility and Corresponding File Extensions*

Option	File type	File extension
`-m`	MPS (Mathematical Programming System)	`.mps`
`-r`	REV (MIPs revise file)	`.rev`
`-s`	SOS (Special Ordered Set)	`.sos`
`-b`	BAS (basis file according to MPS conventions)	`.bas`
`-o`	ORD (priority orders)	`.ord`

Managing Log Files: the Log File Parameter

As ILOG CPLEX is working, it records messages to a log file. By default, it creates the log file in the directory where it is running, and it names the file `cplex.log`. If such a file already exists, ILOG CPLEX adds a line indicating the current time and date and then appends new information to the end of the existing file. That is, it does not overwrite the file, and it distinguishes different sessions within the log file.

You can locate the log file where you like, and you can rename it. Some users, for example, like to create a specifically named log file for each session. Also you can close the log file in case you do not want ILOG CPLEX to record messages to its default log file.

The following sections show you the commands for creating, renaming, relocating, and closing a log file.

Creating, Renaming, Relocating Log Files

◆ In the Interactive Optimizer, use the command `set logfile filename`, substituting the name you prefer for the log file. In other words, use this command to rename or relocate the default log file.

◆ From the Callable Library, first use the routine `CPXfopen()` to open the target file; then use the routine `CPXsetlogfile()`. The *ILOG CPLEX Reference Manual* documents both routines.

Closing Log Files

◆ If you do not want ILOG CPLEX to record messages in a log file, then you can close the log file from the Interactive Optimizer with the command `set logfile *`.

◆ By default, routines from the Callable Library do not write to a log file. However, if you want to close a log file that you created by a call to `CPXsetlogfile()`, call `CPXsetlogfile()` again, and this time, pass a `NULL` pointer as its second argument.

Handling Message Channels: the Output-Channel Parameter

Besides the log-file parameter, ILOG CPLEX offers you output-channel parameters to give you finer control over when and where messages appear in the Interactive Optimizer. Output-channel parameters indicate whether output should or should not appear on screen. They also allow you to designate log files for message channels. The output-channel parameters do not affect the log-file parameter, so it is customary to use the command `set logfile` before the command `set output channel value1 value2`.

In the output-channel command, you can specify a `channel`: `dialog`, `errors`, `logonly`, `results`, and `warnings`. Table 8.2 summarizes the information carried over each channel.

Table 8.2 *Options for the Output-Channel Command*

Channel	Information
`dialog`	messages related to interactive use; e.g., prompts, help messages, greetings
`errors`	messages to inform user that operation could not be performed and why
`logonly`	message to record only in file (not on screen) e.g., multiline messages
`results`	information explicitly requested by user; state, change, progress information
`warnings`	messages to inform user request was performed but unexpected condition may result

The option `value2` lets you specify a file name to redirect output from a channel.

Also in that command, `value1` allows you to turn on or off output to the screen. When `value1` is `y`, output is directed to the screen; when its value is `n`, output is not directed to the screen. Table 8.3 summarizes which channels direct output to the screen by default. If a channel directs output to the screen by default, you can leave `value1` blank to get the same effect as `set output channel y`.

Table 8.3 *Channels Directing Output to Screen or to a File*

Channel	Default value1	Meaning
`dialog`	y	blank directs output to screen but not to a file
`errors`	y	blank directs output to screen and to a file
`logonly`	n	blank directs output only to a file, not to screen

Table 8.3 *Channels Directing Output to Screen or to a File*

Channel	Default value1	Meaning
`results`	`y`	blank directs output to screen and to a file
`warnings`	`y`	blank directs output to screen and to a file

Handling Message Channels: Callable Library Routines

ILOG CPLEX defines several message channels for flexible control over message output:

- `cpxresults` for messages containing status and progress information;
- `cpxerror` for messages issued when a task cannot be completed;
- `cpxwarning` for messages issued when a nonfatal difficulty is encountered; or when an action taken may have side-effects; or when an assumption made may have side-effects;
- `cpxlog` for messages containing information that would not conventionally be displayed on screen but could be useful in a log file.

Output messages flow through message channels to destinations. Message channels are associated with destinations through their destination list. Messages from routines of the ILOG CPLEX Callable Library are assigned internally to one of those predefined channels. Those default channels are C pointers to ILOG CPLEX objects; they are initialized by `CPXopenCPLEX()`; they are *not* global variables. Your application accesses these objects by calling the routine `CPXgetchannels()`. You can use these predefined message channels for your own application messages. You can also define new channels.

An application using routines from the ILOG CPLEX Callable Library produces no output messages unless the application specifies message handling instructions through one or more calls to the message handling routines of the Callable Library. In other words, the destination list of each channel is initially empty.

Messages from multiple channels may be sent to one destination. All predefined ILOG CPLEX channels can be directed to a single file by a call to `CPXsetlogfile()`. Similarly, all predefined ILOG CPLEX channels except `cpxlog` can be directed to the screen by `CPX_PARAM_SCRIND`. For a finer level of control, or to define destinations for application-specific messages, use the following message handling routines, all documented in the *ILOG CPLEX Reference Manual*:

- `CPXmsg()` sets logfile for predefined ILOG CPLEX channels;
- `CPXflushchannel()` flushes a channel to its associated destination;
- `CPXdisconnectchannel()` flushes a channel and clears its destination list;

- `CPXdelchannel()` flushes a channel, clears its destination list, frees memory for that channel;
- `CPXaddchannel()` adds a channel;
- `CPXaddfpdest()` adds a destination file to the list of destinations associated with a channel;
- `CPXdelfpdest()` deletes a destination from the destination list of a channel;
- `CPXaddfuncdest()` adds a destination function to a channel;
- `CPXdelfuncdest()` deletes a destination function to a channel;

Once channel destinations are established, messages can be sent to multiple destinations by a single call to a message-handling routine.

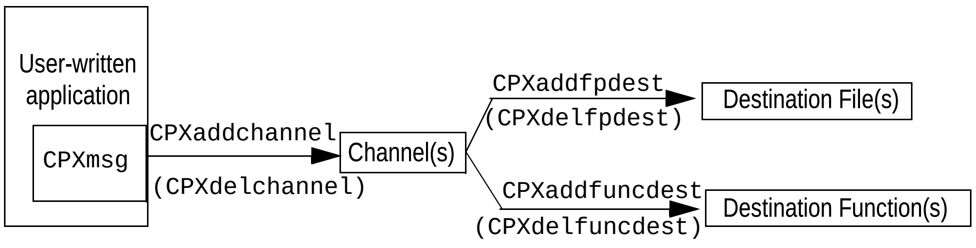

Figure 8.1 *ILOG CPLEX Message Handling Routines*

Example: Using the Message Handler

This example shows you how to use the ILOG CPLEX message handler from the Callable Library. It captures all messages generated by ILOG CPLEX and displays them on screen along with a label indicating which channel sent the message. It also creates a user channel to receive output generated by the program itself. The user channel accepts user-generated messages, displays them on screen with a label, and records them in a file without the label.

This example derives from `lpex1.c`, a program described in the *ILOG CPLEX Getting Started* manual. There are a few differences between the two examples:

- In this example, the function `ourmsgfunc()`—rather than the C functions `printf()` or `fprintf(stderr, . . .)`—manages all output. The program itself or `CPXmsg()` from the ILOG CPLEX Callable Library calls `ourmsgfunc()`. In fact, `CPXmsg()` is a replacement for `printf()`, allowing a message to appear in *more than one place*, for example, both on screen and in a file.

  Only *after* you initialize the ILOG CPLEX environment by calling `CPXopenCPLEX()` can you call `CPXmsg()`. And only *after* you call `CPXgetchannels()` can you use the default ILOG CPLEX channels. Therefore, calls to `ourmsgfunc()` print directly any messages that occur *before* the program gets the address of `cpxerror` (a channel). After a call to `CPXgetchannels()` gets the address of `cpxerror`, and after a call to `CPXaddfuncdest()` associates the message function `ourmsgfunc()` with `cpxerror`, then error messages are generated by calls to `CPXmsg()`.

After the `TERMINATE:` label, any error must be generated with care in case the error message function has not been set up properly. Thus, `ourmsgfunc()` is also called directly to generate any error messages there.

- A call to the ILOG CPLEX Callable Library routine `CPXaddchannel()` initializes the channel `ourchannel`. The C library routine `fopen()` opens the file `lpex5.out` to accept solution information. A call the ILOG CPLEX Callable Library routine `CPXaddfpdest()` associates that file with that channel. Solution information is also displayed on screen since `ourmsgfunc()` is associated with that new channel, too. Thus in the loops near the end of `main()`, when the solution is printed, only one call to `CPXmsg()` suffices to put the output both on screen and into the file. A call to `CPXdelchannel()` deletes `ourchannel`.
- Although `CPXcloseCPLEX()` will automatically delete file- and function-destinations for channels, we recommend that you call `CPXdelfpdest()` and `CPXdelfuncdest()` as the end of your programs.

Complete Program: lpex5.c

The complete program, `lpex5.c`, appears here or online in the standard distribution.

```
#include <ilcplex/cplex.h>

/* Bring in the declarations for the string functions */

#include <string.h>

/* Include declaration for function at end of program */

#ifndef  CPX_PROTOTYPE_MIN

static int
   populatebycolumn  (CPXENVptr env, CPXLPptr lp);

static void CPXPUBLIC
   ourmsgfunc     (void *handle, char *message);


#else

static int
   populatebycolumn ();

static void CPXPUBLIC
   ourmsgfunc     ();

#endif
```

```
/* The problem we are optimizing will have 2 rows, 3 columns
   and 6 nonzeros.  */

#define NUMROWS    2
#define NUMCOLS    3
#define NUMNZ      6

#ifndef  CPX_PROTOTYPE_MIN
int
main (void)
#else
int
main ()
#endif
{
   char     probname[16];  /* Problem name is max 16 characters */

   /* Declare and allocate space for the variables and arrays where we
      will store the optimization results including the status, objective
      value, variable values, dual values, row slacks and variable
      reduced costs. */

   int      solstat;
   double   objval;
   double   x[NUMCOLS];
   double   pi[NUMROWS];
   double   slack[NUMROWS];
   double   dj[NUMCOLS];


   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status;
   int           i, j;
   int           cur_numrows, cur_numcols;
   char          errmsg[1024];

   CPXCHANNELptr  cpxerror   = NULL;
   CPXCHANNELptr  cpxwarning = NULL;
   CPXCHANNELptr  cpxresults = NULL;
   CPXCHANNELptr  ourchannel = NULL;

   char *errorlabel = "cpxerror";
   char *warnlabel  = "cpxwarning";
   char *reslabel   = "cpxresults";
   char *ourlabel   = "Our Channel";

   CPXFILEptr fpout  = NULL;


   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no output,
      so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
      be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

   /* Since the message handler is yet to be set up, we’ll call our
      messaging function directly to print out any errors  */

   if ( env == NULL ) {
      /* The message argument for ourmsgfunc must not be a constant,
         so copy the mesage to a buffer. */
      strcpy (errmsg, "Could not open CPLEX environment.\n");
      ourmsgfunc ("Our Message", errmsg);
      goto TERMINATE;
   }

   /* Now get the standard channels.  If an error, just call our
      message function directly. */

   status = CPXgetchannels (env, &cpxresults, &cpxwarning, &cpxerror, NULL);
   if ( status ) {
      strcpy (errmsg, "Could not get standard channels.\n");
      ourmsgfunc ("Our Message", errmsg);
      CPXgeterrorstring (env, status, errmsg);
      ourmsgfunc ("Our Message", errmsg);
      goto TERMINATE;
   }

   /* Now set up the error channel first.  The label will be "cpxerror" */

   status = CPXaddfuncdest (env, cpxerror, errorlabel, ourmsgfunc);
   if ( status ) {
      strcpy (errmsg, "Could not set up error message handler.\n");
      ourmsgfunc ("Our Message", errmsg);
      CPXgeterrorstring (env, status, errmsg);
      ourmsgfunc ("Our Message", errmsg);
   }

   /* Now that we have the error message handler set up, all CPLEX
      generated errors will go through ourmsgfunc.  So we don’t have
      to use CPXgeterrorstring to determine the text of the message.
      We can also use CPXmsg to do any other printing.  */

   status = CPXaddfuncdest (env, cpxwarning, warnlabel, ourmsgfunc);
   if ( status ) {
      CPXmsg (cpxerror, "Failed to set up handler for cpxwarning.\n");
      goto TERMINATE;
   }

   status = CPXaddfuncdest (env, cpxresults, reslabel, ourmsgfunc);
   if ( status ) {
      CPXmsg (cpxerror, "Failed to set up handler for cpxresults.\n");
      goto TERMINATE;
   }

   /* Now turn on the iteration display. */

   status = CPXsetintparam (env, CPX_PARAM_SIMDISPLAY, 2);
```

```
if ( status ) {
   CPXmsg (cpxerror, "Failed to turn on simplex display level.\n");
   goto TERMINATE;
}

/* Create the problem. */

strcpy (probname, "example");
lp = CPXcreateprob (env, &status, probname);

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.  */

if ( lp == NULL ) {
   CPXmsg (cpxerror, "Failed to create LP.\n");
   goto TERMINATE;
}

/* Now populate the problem with the data. */

status = populatebycolumn (env, lp);

if ( status ) {
   fprintf (stderr, "Failed to populate problem data.\n");
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */

status = CPXlpopt (env, lp);
if ( status ) {
   CPXmsg (cpxerror, "Failed to optimize LP.\n");
   goto TERMINATE;
}

status = CPXsolution (env, lp, &solstat, &objval, x, pi, slack, dj);
if ( status ) {
   CPXmsg (cpxerror, "Failed to obtain solution.\n");
   goto TERMINATE;
}


/* Write the output to the screen.  We will also write it to a
   file as well by setting up a file destination and a function
   destination. */

ourchannel = CPXaddchannel (env);
if ( ourchannel == NULL ) {
   CPXmsg (cpxerror, "Failed to set up our private channel.\n");
   goto TERMINATE;
}

fpout = CPXfopen ("lpex5.msg", "w");
if ( fpout == NULL ) {
      CPXmsg (cpxerror, "Failed to open lpex5.msg file for output.\n");
      goto TERMINATE;
   }
   status = CPXaddfpdest (env, ourchannel, fpout);
   if ( status ) {
      CPXmsg (cpxerror, "Failed to set up output file destination.\n");
      goto TERMINATE;
   }

   status = CPXaddfuncdest (env, ourchannel, ourlabel, ourmsgfunc);
   if ( status ) {
      CPXmsg (cpxerror, "Failed to set up our output function.\n");
      goto TERMINATE;
   }

   /* Now any message to channel ourchannel will go into the file
      and into the file opened above. */

   CPXmsg (ourchannel, "\nSolution status = %d\n", solstat);
   CPXmsg (ourchannel, "Solution value  = %f\n\n", objval);

   /* The size of the problem should be obtained by asking CPLEX what
      the actual size is, rather than using sizes from when the problem
      was built.  cur_numrows and cur_numcols store the current number
      of rows and columns, respectively.  */

   cur_numrows = CPXgetnumrows (env, lp);
   cur_numcols = CPXgetnumcols (env, lp);
   for (i = 0; i < cur_numrows; i++) {
      CPXmsg (ourchannel, "Row %d:  Slack = %10f  Pi = %10f\n",
              i, slack[i], pi[i]);
   }

   for (j = 0; j < cur_numcols; j++) {
      CPXmsg (ourchannel, "Column %d:  Value = %10f  Reduced cost = %10f\n",
              j, x[j], dj[j]);
   }

   /* Finally, write a copy of the problem to a file. */

   status = CPXwriteprob (env, lp, "lpex5.lp", NULL);
   if ( status ) {
      CPXmsg (cpxerror, "Failed to write LP to disk.\n");
      goto TERMINATE;
   }

TERMINATE:

   /* First check if ourchannel is open */

   if ( ourchannel != NULL ) {
      int  chanstat;
      chanstat = CPXdelfuncdest (env, ourchannel, ourlabel, ourmsgfunc);
      if ( chanstat ) {
         strcpy (errmsg, "CPXdelfuncdest failed.\n");
         ourmsgfunc ("Our Message", errmsg);
         if (!status)  status = chanstat;
      }
      if ( fpout != NULL ) {
         chanstat = CPXdelfpdest (env, ourchannel, fpout);
         if ( chanstat ) {
            strcpy (errmsg, "CPXdelfpdest failed.\n");
            ourmsgfunc ("Our Message", errmsg);
            if (!status)  status = chanstat;
         }
         CPXfclose (fpout);
      }

      CPXdelchannel (env, &ourchannel);
   }

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         strcpy (errmsg, "CPXfreeprob failed.\n");
         ourmsgfunc ("Our Message", errmsg);
      }
   }

   /* Now delete our function destinations from the 3 CPLEX channels. */
   if ( cpxresults != NULL ) {
      int  chanstat;
      chanstat = CPXdelfuncdest (env, cpxresults, reslabel, ourmsgfunc);
      if ( chanstat && !status ) {
         status = chanstat;
         strcpy (errmsg, "Failed to delete cpxresults function.\n");
         ourmsgfunc ("Our Message", errmsg);
      }
   }

   if ( cpxwarning != NULL ) {
      int  chanstat;
      chanstat = CPXdelfuncdest (env, cpxwarning, warnlabel, ourmsgfunc);
      if ( chanstat && !status ) {
         status = chanstat;
         strcpy (errmsg, "Failed to delete cpxwarning function.\n");
         ourmsgfunc ("Our Message", errmsg);
      }
   }

   if ( cpxerror != NULL ) {
      int  chanstat;
      chanstat = CPXdelfuncdest (env, cpxerror, errorlabel, ourmsgfunc);
      if ( chanstat && !status ) {
         status = chanstat;
         strcpy (errmsg, "Failed to delete cpxerror function.\n");
         ourmsgfunc ("Our Message", errmsg);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);
      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         strcpy (errmsg, "Could not close CPLEX environment.\n");
         ourmsgfunc ("Our Message", errmsg);
         CPXgeterrorstring (env, status, errmsg);
         ourmsgfunc ("Our Message", errmsg);
      }
   }

   return (status);

}  /* END main */


/* This function builds by column the linear program:

      Maximize
       obj: x1 + 2 x2 + 3 x3
      Subject To
       c1: - x1 + x2 + x3 <= 20
       c2: x1 - 3 x2 + x3 <= 30
      Bounds
       0 <= x1 <= 40
      End
 */

#ifndef  CPX_PROTOTYPE_MIN
static int
populatebycolumn (CPXENVptr env, CPXLPptr lp)
#else
static int
populatebycolumn (env, lp)
CPXENVptr  env;
CPXLPptr   lp;
#endif
{
   int      status    = 0;
   double   obj[NUMCOLS];
   double   lb[NUMCOLS];
   double   ub[NUMCOLS];
   char     *colname[NUMCOLS];
   int      matbeg[NUMCOLS];
   int      matind[NUMNZ];
   double   matval[NUMNZ];
   double   rhs[NUMROWS];
   char     sense[NUMROWS];
   char     *rowname[NUMROWS];

   /* To build the problem by column, create the rows, and then
      add the columns. */

   CPXchgobjsen (env, lp, CPX_MAX);  /* Problem is maximization */
   /* Now create the new rows.  First, populate the arrays. */

   rowname[0] = "c1";
   sense[0]   = ’L’;
   rhs[0]     = 20.0;

   rowname[1] = "c2";
   sense[1]   = ’L’;
   rhs[1]     = 30.0;

   status = CPXnewrows (env, lp, NUMROWS, rhs, sense, NULL, rowname);
   if ( status )   goto TERMINATE;

   /* Now add the new columns.  First, populate the arrays. */

       obj[0] = 1.0;      obj[1] = 2.0;           obj[2] = 3.0;

    matbeg[0] = 0;     matbeg[1] = 2;          matbeg[2] = 4;

    matind[0] = 0;     matind[2] = 0;          matind[4] = 0;
    matval[0] = -1.0;  matval[2] = 1.0;        matval[4] = 1.0;

    matind[1] = 1;     matind[3] = 1;          matind[5] = 1;
    matval[1] = 1.0;   matval[3] = -3.0;       matval[5] = 1.0;

        lb[0] = 0.0;       lb[1] = 0.0;           lb[2]  = 0.0;
        ub[0] = 40.0;      ub[1] = CPX_INFBOUND;  ub[2]  = CPX_INFBOUND;

   colname[0] = "x1"; colname[1] = "x2";      colname[2] = "x3";

   status = CPXaddcols (env, lp, NUMCOLS, NUMNZ, obj, matbeg, matind,
                        matval, lb, ub, colname);
   if ( status )  goto TERMINATE;

TERMINATE:

   return (status);

}  /* END populatebycolumn */


/* For our message functions, we will interpret the handle as a pointer
 * to a string, which will be the label for the channel.  We’ll put
 * angle brackets <> around the message so its clear what the function is
 * sending to us.  We’ll place the newlines that appear at the end of
 * a message after the > bracket.  The ’message’ argument must not be
 * a constant, since it is changed by this function.
 */

#ifndef  CPX_PROTOTYPE_MIN
static void CPXPUBLIC
ourmsgfunc (void *handle, char *message)
#else
static void CPXPUBLIC
ourmsgfunc (handle, message)
void  *handle;
char  *message;
#endif
```

```
{
   char  *label;
   int   lenstr;
   int   flag = 0;

   lenstr = strlen(message);
   if ( message[lenstr-1] == ’\n’ ) {
      message[lenstr-1] = ’\0’;
      flag = 1;
   }

   label = (char *) handle;
   printf ("%-15s: <%s>", label, message);
   if (flag) putchar(’\n’);

   /* If we clobbered the ’\n’, we need to put it back */

   if ( flag )  message[lenstr-1] = ’\n’;

} /* END ourmsgfunc */
```

Using Query Routines

This section tells you how to use query routines. It contains sections on:

- Using Surplus Arguments for Array Allocations
- Example: Using Query Routines

Using Surplus Arguments for Array Allocations

Most of the ILOG CPLEX query routines in the Callable Library require your application to allocate memory for one or more arrays that will contain the results of the query. In many cases, your application—the calling program—does not know the size of these arrays in advance. For example, in a call to `CPXgetcols()` requesting the matrix data for a range of columns, your application needs to pass the arrays `cmatind` and `cmatval` for ILOG CPLEX to populate with matrix coefficients and row indices. However, unless your application has carefully kept track of the number of nonzero columns (that is, the `colnonzero` counts) throughout the problem specification and, if applicable, throughout its modification, the actual length of these arrays remains unknown.

Fortunately, the ILOG CPLEX query routines in the Callable Library contain a *surplus argument* that, when used in conjunction with the array length arguments, enables you first to call the query routine to determine the length of the required array. Then, when the length is known, your application can properly allocate these arrays. Afterwards, your application makes a second call to the query routine with the correct array lengths to obtain the requested data.

For example, consider a program that needs to call `CPXgetcols()` to access a range of columns. Here is the list of arguments for `CPXgetcols()`.

```
CPXgetcols (CPXENVptr env,
            CPXLPptr lp,
            int *nzcnt_p,
            int *cmatbeg,
            int *cmatind,
            double *cmatval,
            int cmatspace,
            int *surplus_p,
            int begin,
            int end);
```

The arrays `cmatind` and `cmatval` require one element for each nonzero matrix coefficient in the requested range of columns. The required length of these arrays, specified in `cmatspace`, remains unknown at the time of the query. Your application—the calling program—can determine the length of these arrays by first calling `CPXgetcols()` with a value of `0` for `cmatspace`. This call will return an error status of `CPXERR_NEGATIVE_SURPLUS` indicating a shortfall of the array length specified in `cmatspace` (in this case, `0`); it will also return the actual number of matrix nonzeros in the requested range of columns. `CPXgetcols()` deposits this shortfall as a negative number in the integer pointed to by `surplus_p`. Your application can then negate this shortfall and allocate the arrays `cmatind` and `cmatval` sufficiently long to contain all the requested matrix elements.

The following sample of code shows you what we mean. The first call to `CPXgetcols()` passes a value of `0` for `cmatspace` in order to obtain the shortfall in `cmatsz`. The sample then uses the shortfall to allocate the arrays `cmatind` and `cmatval` properly; then it calls `CPXgetcols()` again to obtain the actual matrix coefficients and row indices.

```
status = CPXgetcols (env, lp, &nzcnt, cmatbeg, NULL, NULL,
                     0, &cmatsz, 0, numcols - 1);
if ( status != CPXERR_NEGATIVE_SURPLUS ) {
   if ( status != 0 ) {
        CPXmsg (cpxerror,
              “CPXgetcols for surplus failed, status = %d\n”, status);
         goto TERMINATE;
   }
      CPXmsg (cpxwarning, “All columns in range [%d, %d] are empty.\n”,
              0, (numcols - 1));
}
cmatsz   = -cmatsz;
cmatind  = (int *) malloc ((unsigned) (1 + cmatsz)*sizeof(int));
cmatval  = (double *) malloc ((unsigned) (1 + cmatsz)*sizeof(double));
if ( cmatind == NULL || cmatval == NULL ) {
   CPXmsg (cpxerror, “CPXgetcol mallocs failed\n”);
   status = 1;
   goto TERMINATE;
}
status = CPXgetcols (env, lp, &nzcnt, cmatbeg, cmatind, cmatval,
                     cmatsz, &surplus, 0, numcols - 1);
if ( status ) {
   CPXmsg (cpxerror, “CPXgetcols failed, status = %d\n”, status);
   goto TERMINATE;
}
```

That sample code (or your application) does not need to determine the length of the array `cmatbeg`. The array `cmatbeg` has one element for each column in the requested range. Since this length is known ahead of time, your application does not need to call a query routine to calculate it. More generally, query routines use surplus arguments in the way we just described only for the length of any array required to store problem data of unknown length. Problem data in this category includes nonzero matrix entries, row and column names, other problem data names, special ordered sets (SOS), priority orders, and MIP start information.

Example: Using Query Routines

This example uses the ILOG CPLEX Callable Library *query routine* `CPXgetcolname()` to get the column names from a problem object. To do so, it applies the programming pattern we just described in *Using Surplus Arguments for Array Allocations* on page 281. It derives from the example `lpex2.c`, explained in the manual *ILOG CPLEX Getting Started* manual. This query-routine example differs from that simpler example in several ways:

◆ The example calls `CPXgetcolname()` twice after optimization: the first call determines how much space to allocate to hold the names; the second call gets the names and stores them in the arrays `cur_colname` and `cur_colnamestore`.

◆ When the example prints its answer, it uses the names as stored in `cur_colname`. If no names exist there, the example creates fake names.

This example assumes that the current problem has been read from a file by `CPXreadcopyprob()`. You can adapt the example to use other ILOG CPLEX query routines to get information about any problem read from a file.

Complete Program: ilolpex7.cpp

The complete program, `ilolpex7.cpp`, appears here or online in the standard distribution.

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

static void usage (const char *progname);

int
main (int argc, char **argv)
{
   IloEnv env;
   try {
      IloModel model(env);
      IloCplex cplex(env);

      if (( argc != 3 )                             ||
          ( strchr ("podthbn", argv[2][0]) == NULL )  ) {
         usage (argv[0]);
         throw(-1);
      }

      switch (argv[2][0]) {
         case ’o’:
            break;
         case ’p’:
            cplex.setRootAlgorithm(IloCplex::Primal);
            break;
         case ’d’:
            cplex.setRootAlgorithm(IloCplex::Dual);
            break;
         case ’b’:
            cplex.setRootAlgorithm(IloCplex::Barrier);
            cplex.setParam(IloCplex::BarCrossAlg, IloCplex::NoAlg);
            break;
         case ’h’:
            cplex.setRootAlgorithm(IloCplex::Barrier);
            break;
         case ’n’:
            cplex.setRootAlgorithm(IloCplex::NetworkDual);
            break;
         default:
            break;
      }
      IloObjective   obj;
      IloNumVarArray var(env);
      IloRangeArray  rng(env);
      cplex.importModel(model, argv[1], obj, var, rng);

      cplex.extract(model);
      if ( !cplex.solve() ) {
         env.error() << "Failed to optimize LP" << endl;
         throw(-1);
      }

      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;

      for (IloInt i = 0; i < var.getSize(); ++i) {
         if ( var[i].getName() ) env.out() << var[i].getName();
         else                    env.out() << "Fake" << i;
         env.out() << ": " << cplex.getValue(var[i]);
         try {  // basis may not exist
            env.out() << '\t' << cplex.getStatus(var[i]);
         } catch (...) {
         }
         env.out() << endl;
      }
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();
   return 0;
}  // END main


static void usage (const char *progname)
{
   cerr << "Usage: " << progname << " filename algorithm" << endl;
   cerr << "   where filename is a file with extension " << endl;
   cerr << "      MPS, SAV, or LP (lower case is allowed)" << endl;
   cerr << "   and algorithm is one of the letters" << endl;
   cerr << "      o          default" << endl;
   cerr << "      p          primal simplex" << endl;
   cerr << "      d          dual simplex" << endl;
   cerr << "      b          barrier" << endl;
   cerr << "      h          barrier with crossover" << endl;
   cerr << "      n          network simplex" << endl;
   cerr << " Exiting..." << endl;
} // END usage
```

This example uses the ILOG CPLEX Callable Library *query routine* `CPXgetcolname()` to get the column names from a problem object. To do so, it applies the programming pattern we just described in *Using Surplus Arguments for Array Allocations* on page 281. It derives from the example `lpex2.c`, explained in the manual *ILOG CPLEX Getting Started* manual. This query-routine example differs from that simpler example in several ways:

- The example calls `CPXgetcolname()` twice after optimization: the first call determines how much space to allocate to hold the names; the second call gets the names and stores them in the arrays `cur_colname` and `cur_colnamestore`.
- When the example prints its answer, it uses the names as stored in `cur_colname`. If no names exist there, the example creates fake names.

This example assumes that the current problem has been read from a file by `CPXreadcopyprob()`. You can adapt the example to use other ILOG CPLEX query routines to get information about any problem read from a file.

Complete Program: lpex7.c

The complete program, `lpex7.c`, appears here or online in the standard distribution.

```
#include <ilcplex/cplex.h>

/* Bring in the declarations for the string and character functions
   and malloc */

#include <ctype.h>
#include <stdlib.h>
#include <string.h>

/* Include declarations for functions in this program */

#ifndef  CPX_PROTOTYPE_MIN

static void
   free_and_null (char **ptr),
   usage         (char *progname);

#else

static void
   free_and_null (),
   usage         ();

#endif

#ifndef  CPX_PROTOTYPE_MIN
int
main (int argc, char *argv[])
```

```
#else
int
main (argc, argv)
int   argc;
char  *argv[];
#endif
{
   /* Declare and allocate space for the variables and arrays where we will
      store the optimization results including the status, objective value,
      variable values, and basis. */


   int      solstat;
   double   objval;
   double   *x     = NULL;
   int      *cstat = NULL;
   int      *rstat = NULL;


   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status = 0;
   int           j;
   int           cur_numrows, cur_numcols;
   char          **cur_colname = NULL;
   char          *cur_colnamestore = NULL;
   int           cur_colnamespace;
   int           surplus;
   int           method;

   char          *basismsg;

   /* Check the command line arguments */

   if (( argc != 3 )                           ||
       ( strchr ("podhbn", argv[2][0]) == NULL )   ) {
      usage (argv[0]);
      goto TERMINATE;
   }

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no output,
      so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
      be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

   if ( env == NULL ) {
      char  errmsg[1024];
      fprintf (stderr, "Could not open CPLEX environment.\n");
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, "%s", errmsg);
      goto TERMINATE;
   }

   /* Turn on output to the screen */

   status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
   if ( status ) {
      fprintf (stderr,
               "Failure to turn on screen indicator, error %d.\n", status);
      goto TERMINATE;
   }

   /* Create the problem, using the filename as the problem name */

   lp = CPXcreateprob (env, &status, argv[1]);

   /* A returned pointer of NULL may mean that not enough memory
      was available or there was some other problem.  In the case of
      failure, an error message will have been written to the error
      channel from inside CPLEX.  In this example, the setting of
      the parameter CPX_PARAM_SCRIND causes the error message to
      appear on stdout.  Note that most CPLEX routines return
      an error code to indicate the reason for failure.   */

   if ( lp == NULL ) {
      fprintf (stderr, "Failed to create LP.\n");
      goto TERMINATE;
   }

   /* Now read the file, and copy the data into the created lp */

   status = CPXreadcopyprob (env, lp, argv[1], NULL);
   if ( status ) {
      fprintf (stderr, "Failed to read and copy the problem data.\n");
      goto TERMINATE;
   }

   /* Optimize the problem and obtain solution. */

   switch (argv[2][0]) {
      case ’o’:
         method = CPX_ALG_AUTOMATIC;
         break;
      case ’p’:
         method = CPX_ALG_PRIMAL;
         break;
      case ’d’:
         method = CPX_ALG_DUAL;
         break;
      case 'n':
         method = CPX_ALG_NET;
         break;
      case 'h':
         method = CPX_ALG_BARRIER;
         break;
      case 'b':
         method = CPX_ALG_BARRIER;
         status = CPXsetintparam (env, CPX_PARAM_BARCROSSALG, CPX_ALG_NONE);
         if ( status ) {
            fprintf (stderr,
                     "Failed to set the crossover method, error %d.\n",
status);
            goto TERMINATE;
         }
         break;
      default:
         method = CPX_ALG_NONE;
         break;
   }

   status = CPXsetintparam (env, CPX_PARAM_LPMETHOD, method);
   if ( status ) {
      fprintf (stderr,
               "Failed to set the optimization method, error %d.\n", status);
      goto TERMINATE;
   }

   status = CPXlpopt (env, lp);
   if ( status ) {
      fprintf (stderr, "Failed to optimize LP.\n");
      goto TERMINATE;
   }

   solstat = CPXgetstat (env, lp);
   status  = CPXgetobjval (env, lp, &objval);

   if ( status ) {
      fprintf (stderr,"Failed to obtain objective value.\n");
      goto TERMINATE;
   }

   printf ("Solution status %d.  Objective value %.10g\n",
           solstat, objval);

   /* The size of the problem should be obtained by asking CPLEX what
      the actual size is.  cur_numrows and cur_numcols store the
      current number of rows and columns, respectively.  */

   cur_numcols = CPXgetnumcols (env, lp);
   cur_numrows = CPXgetnumrows (env, lp);

   /* Allocate space for basis and solution */
```

```
cstat = (int *)    malloc (cur_numcols*sizeof(int));
rstat = (int *)    malloc (cur_numrows*sizeof(int));
x     = (double *) malloc (cur_numcols*sizeof(double));

if ( cstat == NULL || rstat == NULL || x == NULL ) {
   fprintf (stderr,"No memory for basis statuses.\n");
   goto TERMINATE;
}

/* If CPXgetbase causes an error, we don’t want to see that error
   message on the screen.  So turn off the screen indicator for
   this call, and turn it back on afterwards.  */

CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_OFF);
status = CPXgetbase (env, lp, cstat, rstat);
CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);

if ( status == CPXERR_NO_BASIS ) {
   printf ("No basis exists.\n");
   free_and_null ((char **) &cstat);
   free_and_null ((char **) &rstat);
}
else if ( status ) {
   fprintf (stderr,"Failed to get basis. error %d.\n", status);
   goto TERMINATE;
}

status = CPXgetx (env, lp, x, 0, cur_numcols-1);
if ( status ) {
   fprintf (stderr, "Failed to obtain primal solution.\n");
   goto TERMINATE;
}

/* Now get the column names for the problem.  First we determine how
   much space is used to hold the names, and then do the allocation.
   Then we call CPXgetcolname() to get the actual names. */

status = CPXgetcolname (env, lp, NULL, NULL, 0, &surplus, 0,
                        cur_numcols-1);

if (( status != CPXERR_NEGATIVE_SURPLUS ) &&
    ( status != 0 )                         )  {
   fprintf (stderr,
            "Could not determine amount of space for column names.\n");
   goto TERMINATE;
}

cur_colnamespace = - surplus;
if ( cur_colnamespace > 0 ) {
   cur_colname      = (char **) malloc (sizeof(char *)*cur_numcols);
   cur_colnamestore = (char *)  malloc (cur_colnamespace);
   if ( cur_colname      == NULL ||
           cur_colnamestore == NULL   ) {
         fprintf (stderr, "Failed to get memory for column names.\n");
         status = -1;
         goto TERMINATE;
      }
      status = CPXgetcolname (env, lp, cur_colname, cur_colnamestore,
                              cur_colnamespace, &surplus, 0, cur_numcols-1);
      if ( status ) {
         fprintf (stderr, "CPXgetcolname failed.\n");
         goto TERMINATE;
      }
   }
   else {
      printf ("No names associated with problem.  Using Fake names.\n");
   }

   /* Write out the solution */

   for (j = 0; j < cur_numcols; j++) {
      if ( cur_colnamespace > 0 ) {
         printf ("%-16s:  ", cur_colname[j]);
      }
      else {
         printf ("Fake%-6.6d      :  ", j);;
      }
      printf ("%17.10g", x[j]);
      if ( cstat != NULL ) {
         switch (cstat[j]) {
            case CPX_AT_LOWER:
               basismsg = "Nonbasic at lower bound";
               break;
            case CPX_BASIC:
               basismsg = "Basic";
               break;
            case CPX_AT_UPPER:
               basismsg = "Nonbasic at upper bound";
               break;
            case CPX_FREE_SUPER:
               basismsg = "Superbasic, or free variable at zero";
               break;
            default:
               basismsg = "Bad basis status";
               break;
         }
         printf ("  %s",basismsg);
      }
      printf ("\n");
   }


TERMINATE:
   /* Free up the basis and solution */

   free_and_null ((char **) &cstat);
   free_and_null ((char **) &rstat);
   free_and_null ((char **) &x);
   free_and_null ((char **) &cur_colname);
   free_and_null ((char **) &cur_colnamestore);

   /* Free up the problem, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   return (status);

}  /* END main */



/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

#ifndef  CPX_PROTOTYPE_MIN
static void
free_and_null (char **ptr)
#else
static void
free_and_null (ptr)
char  **ptr;
#endif
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */


#ifndef  CPX_PROTOTYPE_MIN
static void
usage (char *progname)
#else
static void
usage (progname)
char *progname;
#endif
{
   fprintf (stderr,"Usage: %s filename algorithm\n", progname);
   fprintf (stderr,"   where filename is a file with extension \n");
   fprintf (stderr,"      MPS, SAV, or LP (lower case is allowed)\n");
   fprintf (stderr,"   and algorithm is one of the letters\n");
   fprintf (stderr,"      o          default\n");
   fprintf (stderr,"      p          primal simplex\n");
   fprintf (stderr,"      d          dual simplex\n");
   fprintf (stderr,"      n          network simplex\n");
   fprintf (stderr,"      b          barrier\n");
   fprintf (stderr,"      h          barrier with crossover\n");
   fprintf (stderr," Exiting...\n");
} /* END usage */
```

Using Callbacks

This section introduces the topic of callback routines, which allow you to closely monitor and guide the behavior of CPLEX optimizers. It includes information on:

- Diagnostic Callbacks
- Control Callbacks for `IloCplex`

CPLEX callbacks allow user code to be executed regularly during an optimization. There are two types of callbacks, diagnostic callbacks and control callbacks, which are discussed separately in the following sections. To use callbacks with CPLEX, you must first write the callback function, and then pass it to CPLEX.

Diagnostic Callbacks

Diagnostic callbacks allow you to monitor an ongoing optimization, and optionally abort it. These callbacks are distinguished by the place where they are called during an optimization. There are 10 such places where diagnostic callbacks are called:

- The presolve `callback` is called regularly during presolve.
- The crossover `callback` is called regularly during crossover from a barrier solution to a SIMPLEX basis.
- The network `callback` is called regularly during the network simplex.
- The barrier `callback` is called at each iteration during the barrier algorithm.
- The primal `callback` is called at each iteration during the primal simplex algorithm.
- The dual `callback` is called at each iteration during the dual simplex algorithm.
- The MIP `callback` is called at each node during the branch & cut search.
- The probing `callback` is called regularly during probing.
- The fractional cut `callback` is called regularly during the separation for fractional cuts.
- The disjunctive cut `callback` is called regularly during the separation for disjunctive cuts.

Implementing Callbacks In CPLEX with Concert Technology

With `IloCplex`, callbacks are accessed via a the `IloCplex::Callback` handle class. It points to an implementation object of a subclass of `IloCplex::CallbackI`. One such implementation class is provided for each type of callback. The implementation class provides the functions that can be used for the particular callback as protected member functions. To reflect the fact that some callbacks share part of their protected API, the callback classes are organized in a class hierarchy as shown by this diagram:

```
IloCplex::CallbackI
   |
   +--- IloCplex::PresolveCallbackI
   |
   +--- IloCplex::CrossoverCallbackI
   |
   +--- IloCplex::NetworkCallbackI
   |
   +--- IloCplex::LPCallbackI
   |       |
   |       +--- IloCplex::BarrierCallbackI
   |       |
   |       +--- IloCplex::PrimalSimplexCallbackI
   |       |
   |       +--- IloCplex::DualSimplexCallbackI
   |
   +--- IloCplex::MIPCallbackI
           |
           +--- IloCplex::ProbingCallback
           |
           +--- IloCplex::FractionalCutCallbackI
           |
           +--- IloCplex::DisjunctiveCutCallbackI
```

This means that, for example, all functions available for the MIP callback are also available for the probing, fractional cut, and disjunctive cut callbacks. In particular, the function to abort the current optimization is provided by the class `IloCplex::CallbackI` and is thus available to all callbacks.

There are two ways of implementing callbacks for `IloCplex`: a more complex way that exposes all the C++ implementation details, and a simplified way that uses macros to handle the C++ technicalities. We will first expose the more complex way and discuss the underlying design. To quickly implement your callback without details on the internal design, proceed directly to *Writing Callbacks with Macros* on page 296.

Writing Callback Classes by Hand

To implement your own callback for `IloCplex`, first select the callback class corresponding to the callback you want implemented. From it derive your own implementation class and overwrite the virtual method `main()`. This is where you implement the callback actions, using the protected member functions of the callback class from which you derived your callback or one of its base classes.

Next write a function that creates a new object of your implementation class using the environment operator `new` and returning it as an `IloCplex::Callback` handle object. Here is an example implementation of such a function:

```
IloCplex::Callback MyCallback(IloEnv env, IloInt num) {
return (new (env) MyCallbackI(num));
}
```

Once the implementation is completed, use it with `IloCplex` by calling `cplex.use()` with the handle object returned by your callback function. To remove a callback that is being used by a `cplex` object, call `callback.end()` on the `IloCplex::Callback` handle callback.

One object of a callback implementation class can be used with only one `IloCplex` object at a time. Thus, when you use a callback with more than one `cplex` object, a copy of the implementation object is created every time `cplex.use()` is called except for the first time. Method `IloCplex::use()` returns a handle to the callback object that has actually been installed to enable calling `end()` on it.

To construct the copies of the callback objects, class `IloCplex::CallbackI` defines another pure virtual method:

```
virtual IloCplex::CallbackI* IloCplex::CallbackI::makeClone()
     const = 0;
```

which must be implemented for your callback class. This method will be called to create the copies needed for using a callback on different `cplex` objects or on one `cplex` object with a parallel optimizer.

In most cases you can avoid writing callback classes by hand, using supplied macros that make the process as easy as implementing a function. You must implement a callback by hand only if the callback manages internal data not passed as arguments, or if the callback requires eight or more parameters.

Writing Callbacks with Macros

Here is how to implement a callback using macros. First, determine which callback you want to implement and how many arguments to pass to the callback function. These two pieces of information determine the macro you need to use.

For example, to implement a dual simplex callback with one parameter, the macro is `ILODUALSIMPLEXCALLBACK1`. Generally, for every callback type `XXX` and any number of parameters `n` from `0` to `7` there is a macro called `ILOXXXCALLBACKn`. The following table lists the callbacks and the corresponding macros and classes (where n is a placeholder for 0..7):

Table 8.4 *Callback Macros*

Callback	Macro	Class
presolve	`ILOPRESOLVECALLBACKn`	`IloCplex::PresolveCallbackI`
LP	`ILOLPCALLBACKn`	`IloCplex::LPCallbackI`
primal simplex	`ILOPRIMALSIMPLEXCALLBACKn`	`IloCplex::PrimalSimpleXCallbackI`
dual simplex	`ILODUALSIMPLEXCALLBACKn`	`IloCplex::DualSimpleXCallbackI`

***Table 8.4** Callback Macros (Continued)*

Callback	Macro	Class
barrier	`ILOBARRIERCALLBACKn`	`IloCplex::BarrierCallbackI`
crossover	`ILOCROSSOVERCALLBACKn`	`IloCplex::CrossoverCallbackI`
network	`ILONETWORKCALLBACKn`	`IloCplex::NetworkCallbackI`
MIP	`ILOMIPCALLBACKn`	`IloCplex::MIPCallbackI`
probing	`ILOPROBINGCALLBACKn`	`IloCplex::ProbingCallbackI`
fractional cut	`ILOFRACTIONALCUTCALLBACKn`	`IloCplex::FractionalCutCallbackI`
disjunctive cut	`ILODISJUNCTIVECUTCALLBACKn`	`IloCplex::DisjunctiveCutCallbackI`

The protected member functions of the corresponding class and its base classes determine the functions that can be called for implementing your callback (see the *ILOG CPLEX Reference Manual*).

Here is an example of how to implement a dual simplex callback with the name `MyCallback` that takes one parameter:

```
ILODUALSIMPLEXCALLBACK1(MyCallback, IloInt, num) {
  if ( getNiterations() == num ) abort();
}
```

This callback aborts the dual simplex algorithm at the `num`th iteration. It queries the current iteration number by calling function `getNiterations()`, which is a protected member function of class `IloCplex::LPCallbackI`.

To use this callback with an `IloCplex` object `cplex`, simply call:

```
IloCplex::Callback mycallback = cplex.use(MyCallback(env, 10));
```

The callback that is added to `cplex` is returned by the method `use` and stored in variable `mycallback`. This allows you to call `mycallback.end()`to remove the callback from `cplex`. If you do not intend accessing your callback, for example in order to delete it before ending the environment, you may safely leave out the declaration and initialization of variable `mycallback`.

Callback Interface

Two callback classes in the hierarchy need extra attention. The first is the base class `IloCplex::CallbackI`. Since there is no corresponding callback in CPLEX, this class cannot be used for implementing user callbacks. Instead, its purpose is to provide an interface common to all callback functions. This consists of the methods `getModel()`, which returns the model that is extracted to the CPLEX object that is calling the callback, `getEnv()`, which returns the corresponding environment, and `abort()`, which aborts the current optimization. Further, methods `getNrows()` and `getNcols()` allow you to query the number of rows and columns of the current `cplex` LP matrix. These methods can be called from all callbacks.

> ***Note:*** *No manipulation of the model or, more precisely, any extracted modeling object is allowed during the execution of a callback. If you want to use your callback with a parallel optimizer, no modification is allowed of any array or expression not local to the callback function itself (that is, constructed and* `end()`*ed in it). The only exception is the modification of array elements. For example,* `x[i] = 0` *would be permissible, whereas* `x.add(0)` *would not unless* `x` *is a local array of the callback. To avoid any problems when changing from a sequential optimizer to a parallel one, it is advisable to always observe this restriction.*

The LP Callback

The second special callback class is `IloCplex::LPCallbackI`. If you create an LP callback and use it with an `IloCplex` object, this callback will be used for all of the barrier, dual simplex, and primal simplex callbacks. In other words, implementing and using one LP callback is equivalent to writing and using these three callbacks independently.

Example: Deriving the Primal Simplex Callback

This example demonstrates the use of the primal simplex callback to print logging information at each iteration. It is a modification of example `ilolpex1.cpp`, so we will restrict our discussion to the differences. The following code:

```
ILOPRIMALSIMPLEXCALLBACKI0(MyCallback) {
  cout << "Iteration " << getNiterations() << ": ";
    if ( isFeasible() ) {
    cout << "Objective = " << getObjValue() << endl;
  }
  else {
    cout << "Infeasibility measure = " << getInfeasibility() << endl;
  }
}
```

defines the callback `MyCallback` without parameters with the code enclosed in the outer `{}`.

The callback prints the iteration number to `cout`. Then, depending on whether the current solution is feasible or not, it prints the objective value or infeasibility measure to `cout`. The functions `getNiterations()`, `isFeasible()`, `getObjValue()`, and `getInfeasibility()` are member functions provided in the callback's base `class IloCplex::PrimalSimplexCallbackI`. See the *ILOG CPLEX Reference Manual* for the complete list of methods provided for each callback class.

Here is how the macro `ILOPRIMALSIMPLEXCALLBACK0` is expanded:

```
class MyCallbackI : public IloCplex::PrimalSimplexCallbackI {
  void main();
  IloCplex::CallbackI* makeClone() const {
    return (new (getEnv()) MyCallbackI(*this));
  }
};
IloCplex::Callback MyCallback(IloEnv env) {
  return (IloCplex::Callback(new (env) MyCallbackI()));
}
void MyCallbackI::main() {
  cout << "Iteration " << getNiterations() << ": ";
  if ( isFeasible() ) {
    cout << "Objective = " << getObjValue() << endl;
  }
  else {
     cout << "Infeasibility measure = " << getInfeasibility() << endl;
  }
}
```

The `0` in the macro indicates that `0` parameters are passed to the constructor of the callback. For callbacks requiring up to 7 parameters similar macros are defined where the `0` is replaced by the number of parameters, ranging from 1 through 7. For an example of this using the cut callback, see *Example: Controlling Cuts* on page 312. If you need more than 7 parameters, you will need to derive your callback class yourself without the help of a macro.

After the callback `MyCallback` is defined, it can be used with the line:

```
   cplex.use(MyCallback(env));
```

Function `MyCallback` creates an instance of the implementation class `MyCallbackI`. A handle to this implementation object is passed to `cplex` method `use()`.

If your application defines more than one primal simplex callback object (possibly with different subclasses), only the last one passed to CPLEX with the `use` method is actually used during primal simplex. On the other hand, `IloCplex` can handle one callback for each callback class at the same time. For example a primal simplex callback and a MIP callback can be used at the same time.

Complete Program: ilolpex4.cpp

The complete program, `ilolpex4.cpp`, appears here or online in the standard distribution.

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


ILOPRIMALSIMPLEXCALLBACK0(MyCallback) {
  cout << "Iteration " << getNiterations() << ": ";
  if ( isFeasible() ) {
     cout << "Objective = " << getObjValue() << endl;
  } else {
     cout << "Infeasibility measure = " << getInfeasibility() << endl;
  }
}


static void
   populatebycolumn (IloModel model, IloNumVarArray var, IloRangeArray rng);

int
main (int argc, char **argv)
{
   IloEnv env;
   try {
      IloModel model(env, "example");

      IloNumVarArray var(env);
      IloRangeArray  rng(env);
      populatebycolumn (model, var, rng);

      IloCplex cplex(model);
      cplex.setOut(env.getNullStream());
      cplex.setRootAlgorithm(IloCplex::Primal);
      cplex.use(MyCallback(env));
      cplex.solve();

      cplex.out() << "Solution status = " << cplex.getStatus() << endl;
      cplex.out() << "Solution value  = " << cplex.getObjValue() << endl;

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      env.out() << "Values        = " << vals << endl;
      cplex.getSlacks(vals, rng);
      env.out() << "Slacks        = " << vals << endl;
      cplex.getDuals(vals, rng);
      env.out() << "Duals         = " << vals << endl;
      cplex.getReducedCosts(vals, var);
      env.out() << "Reduced Costs = " << vals << endl;

      cplex.exportModel("lpex4.lp");
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();
   return 0;
}  // END main


// To populate by column, we first create the rows, and then add the
// columns.
```

```
static void
populatebycolumn (IloModel model, IloNumVarArray x, IloRangeArray c)
{
   IloEnv env = model.getEnv();

   IloObjective obj = IloMaximize(env);
   c.add(IloRange(env, -IloInfinity, 20.0));
   c.add(IloRange(env, -IloInfinity, 30.0));

   x.add(IloNumVar(obj(1.0) + c[0](-1.0) + c[1]( 1.0), 35.0, 40.0));
   x.add(obj(2.0) + c[0]( 1.0) + c[1](-3.0));
   x.add(obj(3.0) + c[0]( 1.0) + c[1]( 1.0));

   model.add(obj);
   model.add(c);

}  // END populatebycolumn
```

Implementing Callbacks in the Callable C Library

ILOG CPLEX optimization routines in the Callable Library incorporate a callback facility to allow your application to transfer control temporarily from ILOG CPLEX to the calling application. Using callbacks, your application can implement interrupt capability, for example, or create displays of optimization progress. Once control is transferred back to a function in the calling application, the calling application can retrieve specific information about the current optimization from the routine `CPXgetcallbackinfo()`. Optionally, the calling application can then tell ILOG CPLEX to discontinue optimization.

To implement and use a callback in your application, you must first write the callback function and then tell ILOG CPLEX about it. For more information about the ILOG CPLEX Callable Library routines for callbacks, see the *ILOG CPLEX Reference Manual*.

Setting Callbacks

In the Callable Library, control callbacks are grouped into two groups: LP callbacks and MIP callbacks. For each group, one callback function can be set, by calling functions `CPXsetlpcallbackfunc()` and `CPXsetmipcallbackfunc()`, respectively. The function `CPXsetlpcallbackfunc()` is called for callbacks 1 through 6, while the function `CPXsetmipcallbackfunc()` is called for callbacks 7 through 10. You can distinguish between the actual callbacks by querying the parameter `wherefrom` which is passed to the callback function as parameter by CPLEX.

Callbacks for LPs and for MIPs

ILOG CPLEX will evaluate two user-defined callback functions, one during the solution of LP problems and one during the solution of MIP problems (if you are licensed to use the MIP optimizer). ILOG CPLEX calls the LP callback once per iteration during the solution of an LP problem and periodically during the presolve of LP and MIP preprocessing.

ILOG CPLEX calls the MIP callback once before each subproblem is solved in the branch & cut process.

Every user-defined callback must have these arguments:

- `env`, a pointer to the ILOG CPLEX environment;
- `cbdata`, a pointer to ILOG CPLEX internal data structures needed by `CPXgetcallbackinfo()`;
- `wherefrom`, indicates which optimizer is calling the callback;
- `cbhandle`, a pointer supplied when your application calls `CPXsetlpcallbackfunc()` or `CPXsetmipcallbackfunc()` (so that the callback has access to private user data).

The arguments `wherefrom` and `cbhandle` should be used only in calls to `CPXgetcallbackinfo()`.

Return Values for Callbacks

A user-written callback should return a nonzero value if the user wishes to stop the optimization and a value of zero otherwise.

For LP problems, if the callback returns a nonzero value, the solution process will terminate. If the process was not terminated during the presolve process, the status returned by the function `IloCplex::getStatus` or the routines `CPXsolution()` or `CPXgetstat()` will be one of the values in Table 8.5.

Table 8.5 *Status of nonzero callbacks for LPs*

Value	Symbolic constant	Meaning
12	CPX_ABORT_FEAS	aborted in Phase II (simplex)
13	CPX_ABORT_INFEAS	aborted in Phase I (simplex)
14	CPX_ABORT_DUAL_INFEAS	primal feasible, dual infeasible (barrier)
15	CPX_ABORT_PRIM_INFEAS	primal infeasible, dual feasible (barrier)
16	CPX_ABORT_PRIM_DUAL_INFEAS	primal and dual both infeasible (barrier)
17	CPX_ABORT_PRIM_DUAL_FEAS	primal and dual both feasible (barrier)
18	CPX_ABORT_CROSSOVER	aborted in crossover (barrier)

For both LP and MIP problems, if the LP callback returns a nonzero value during presolve preprocessing, the optimizer will return the value `CPXERR_PRESLV_ABORT`, and no solution information will be available.

For MIP problems, if the callback returns a nonzero value, the solution process will terminate and the status returned by `IloCplex::getStatus()` or `CPXgetstat()` will be one of the values in Table 8.6.

Table 8.6 *Status of nonzero callbacks for MIPs*

Value	Symbolic constant	Meaning
113	CPXMIP_ABORT_FEAS	current solution integer feasible
114	CPXMIP_ABORT_INFEAS	no integer feasible solution found

Interaction Between Callbacks and CPLEX Parallel Optimizers

When you use callback routines, and invoke the parallel version of CPLEX optimizers, you need to be aware that the CPLEX environment passed to the callback routine corresponds to an individual CPLEX thread rather than to the original environment created. CPLEX frees this environment when finished with the thread. This does not affect most uses of the callback function. However, keep in mind that CPLEX associates problem objects, parameter settings, and message channels with the environment that specifies them. CPLEX therefore frees these items when it removes that environment; if the callback uses routines like `CPXcreateprob`, `CPXcloneprob` or `CPXgetchannels`, those objects remain allocated only as long as the associated environment does. Similarly, setting parameters with routines like `CPXsetintparam` affects settings only within the thread. So, applications that access CPLEX objects in the callback should use the original environment you created by if they need to access these objects outside the scope of the callback function.

Example: Using Callbacks

This example shows you how to use callbacks effectively with routines from the ILOG CPLEX Callable Library. It is based on `lpex1.c`, a program described in the manual *Getting Started with ILOG CPLEX*. This example about callbacks differs from that simpler one in several ways:

◆ To make the output more interesting, this example optimizes a slightly different linear program.

◆ The ILOG CPLEX screen indicator (that is, the parameter `CPX_PARAM_SCRIND`) is not turned on. Only the callback function produces output. Consequently, this program calls `CPXgeterrorstring()` to determine any error messages and then prints them. After the `TERMINATE:` label, the program uses separate status variables so that if an error occurred earlier, its error status will not be lost or destroyed by freeing the problem object and closing the ILOG CPLEX environment. Table 8.7 summarizes those status variables.

Table 8.7 *Status Variables in* `lpex4.c`

Variable	Represents status returned by this routine
`frstatus`	`CPXfreeprob()`
`clstatus`	`CPXcloseCPLEX()`

◆ The function `mycallback()` at the end of the program is called by the optimizer. This function tests whether the primal simplex optimizer has been called. If so, then a call to `CPXgetcallbackinfo()` gets the following information:

- iteration count;
- feasibility indicator;
- sum of infeasibilities (if infeasible);
- objective value (if feasible).

The function then prints these values to indicate progress.

◆ Before the program calls `CPXlpopt()`, the default optimizer from the ILOG CPLEX Callable Library, it sets the callback function by calling `CPXsetlpcallbackfunc()`. It unsets the callback immediately after optimization.

This callback function offers a model for graphic user interfaces that display optimization progress as well as those GUIs that allow a user to interrupt and stop optimization. If you want to provide your end-user a facility like that to interrupt and stop optimization, then you should make `mycallback()` return a nonzero value to indicate the end-user interrupt.

Complete Program: lpex4.c

The complete program, `lpex4.c`, appears here or online in the standard distribution.

```
#include <ilcplex/cplex.h>

/* Bring in the declarations for the string functions */

#include <string.h>

/* Include declaration for function at end of program */

#ifndef  CPX_PROTOTYPE_MIN

static int
   populatebycolumn  (CPXENVptr env, CPXLPptr lp);

static int CPXPUBLIC
```

```
   mycallback (CPXENVptr env, void *cbdata, int wherefrom,
               void *cbhandle);

#else

static int
   populatebycolumn ();

static int CPXPUBLIC
   mycallback ();


#endif


/* The problem we are optimizing will have 2 rows, 3 columns
   and 6 nonzeros.  */

#define NUMROWS    2
#define NUMCOLS    3
#define NUMNZ      6

#ifndef  CPX_PROTOTYPE_MIN
int
main (void)
#else
int
main ()
#endif
{
   char     probname[16];  /* Problem name is max 16 characters */

   /* Declare and allocate space for the variables and arrays where we
      will store the optimization results including the status, objective
      value, variable values, dual values, row slacks and variable
      reduced costs. */

   int      solstat;
   double   objval;
   double   x[NUMCOLS];
   double   pi[NUMROWS];
   double   slack[NUMROWS];
   double   dj[NUMCOLS];


   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status;
   int           i, j;
   int           cur_numrows, cur_numcols;

   /* Initialize the CPLEX environment */
```

```
env = CPXopenCPLEX (&status);

/* If an error occurs, the status value indicates the reason for
   failure.  The error message will be printed at the end of the
   program. */

if ( env == NULL ) {
   fprintf (stderr, “Could not open CPLEX environment.\n”);
   goto TERMINATE;
}

/* Turn *off* output to the screen since we’ll be producing it
   via the callback function.  This also means we won’t see any
   CPLEX generated errors, but we’ll handle that at the end of
   the program. */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_OFF);
if ( status ) {
   fprintf (stderr,
            “Failure to turn off screen indicator, error %d.\n”, status);
   goto TERMINATE;
}

/* Create the problem. */

strcpy (probname, “example”);
lp = CPXcreateprob (env, &status, probname);

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, we wouldn’t see
   an error message from CPXcreateprob since we turned off the
   CPX_PARAM_SCRIND parameter above.  The only way to see this message
   would be to use the CPLEX message handler, but that clutters up
   the simplicity of this example, which has a point of illustrating
   the CPLEX callback functionality.   */

if ( lp == NULL ) {
   fprintf (stderr, “Failed to create LP.\n”);
   goto TERMINATE;
}

/* Now populate the problem with the data. */

status = populatebycolumn (env, lp);

if ( status ) {
   fprintf (stderr, “Failed to populate problem data.\n”);
   goto TERMINATE;
}
```

```
   status = CPXsetlpcallbackfunc (env, mycallback, NULL);
   if ( status ) {
      fprintf (stderr, “Failed to set callback function.\n”);
      goto TERMINATE;
   }

   /* Optimize the problem and obtain solution. */

   status = CPXsetintparam (env, CPX_PARAM_LPMETHOD, CPX_ALG_PRIMAL);
   if ( status ) {
      fprintf (stderr,
               “Failed to set the optimization method, error %d.\n”, status);
      goto TERMINATE;
   }

   status = CPXlpopt (env, lp);
   if ( status ) {
      fprintf (stderr, “Failed to optimize LP.\n”);
      goto TERMINATE;
   }

   /* Turn off the callback function.  This isn’t strictly necessary,
      but is good practice.  Note that the cast in front of NULL
      is only necessary for some compilers.   */

#ifndef CPX_PROTOTYPE_MIN
   status = CPXsetlpcallbackfunc (env,
              (int (CPXPUBLIC *)(CPXENVptr, void *, int, void *)) NULL, NULL);
#else
   status = CPXsetlpcallbackfunc (env, (int (CPXPUBLIC *)()) NULL, NULL);
#endif
   if ( status ) {
      fprintf (stderr, “Failed to turn off callback function.\n”);
      goto TERMINATE;
   }

   status = CPXsolution (env, lp, &solstat, &objval, x, pi, slack, dj);
   if ( status ) {
      fprintf (stderr, “Failed to obtain solution.\n”);
      goto TERMINATE;
   }


   /* Write the output to the screen. */

   printf (“\nSolution status = %d\n”, solstat);
   printf (“Solution value  = %f\n\n”, objval);

   /* The size of the problem should be obtained by asking CPLEX what
      the actual size is, rather than using sizes from when the problem
      was built.  cur_numrows and cur_numcols store the current number
      of rows and columns, respectively.  */
   cur_numrows = CPXgetnumrows (env, lp);
   cur_numcols = CPXgetnumcols (env, lp);
   for (i = 0; i < cur_numrows; i++) {
      printf (“Row %d:  Slack = %10f  Pi = %10f\n”, i, slack[i], pi[i]);
   }

   for (j = 0; j < cur_numcols; j++) {
      printf (“Column %d:  Value = %10f  Reduced cost = %10f\n”,
              j, x[j], dj[j]);
   }

   /* Finally, write a copy of the problem to a file. */

   status = CPXwriteprob (env, lp, “lpex4.lp”, NULL);
   if ( status ) {
      fprintf (stderr, “Failed to write LP to disk.\n”);
      goto TERMINATE;
   }

TERMINATE:

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      int  frstatus;
      frstatus = CPXfreeprob (env, &lp);
      if ( frstatus ) {
         fprintf (stderr, “CPXfreeprob failed, error code %d.\n”, frstatus);
         if (( !status ) && frstatus )  status = frstatus;
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      int  clstatus;
      clstatus = CPXcloseCPLEX (&env);

      if ( clstatus ) {
         fprintf (stderr, “CPXcloseCPLEX failed, error code %d.\n”, clstatus);
         if (( !status ) && clstatus )  status = clstatus;
      }
   }

   if ( status ) {
      char  errmsg[1024];

      /* Note that since we have turned off the CPLEX screen indicator,
         we’ll need to print the error message ourselves. */

      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, “%s”, errmsg);
   }

   return (status);

}  /* END main */


/* This function builds by column the linear program:

      Maximize
       obj: x1 + 2 x2 + 3 x3
      Subject To
       c1: - x1 + x2 + x3 <= 20
       c2: x1 - 3 x2 + x3 <= 30
      Bounds
       35 <= x1 <= 40
      End
 */

#ifndef  CPX_PROTOTYPE_MIN
static int
populatebycolumn (CPXENVptr env, CPXLPptr lp)
#else
static int
populatebycolumn (env, lp)
CPXENVptr  env;
CPXLPptr   lp;
#endif
{
   int      status    = 0;
   double   obj[NUMCOLS];
   double   lb[NUMCOLS];
   double   ub[NUMCOLS];
   char     *colname[NUMCOLS];
   int      matbeg[NUMCOLS];
   int      matind[NUMNZ];
   double   matval[NUMNZ];
   double   rhs[NUMROWS];
   char     sense[NUMROWS];
   char     *rowname[NUMROWS];

   /* To build the problem by column, create the rows, and then
      add the columns. */

   CPXchgobjsen (env, lp, CPX_MAX);  /* Problem is maximization */

   /* Now create the new rows.  First, populate the arrays. */

   rowname[0] = “c1”;
   sense[0]   = ‘L’;
   rhs[0]     = 20.0;
   rowname[1] = “c2”;
   sense[1]   = ‘L’;
   rhs[1]     = 30.0;

   status = CPXnewrows (env, lp, NUMROWS, rhs, sense, NULL, rowname);
   if ( status )   goto TERMINATE;

   /* Now add the new columns.  First, populate the arrays. */

       obj[0] = 1.0;      obj[1] = 2.0;           obj[2] = 3.0;

    matbeg[0] = 0;     matbeg[1] = 2;          matbeg[2] = 4;

    matind[0] = 0;     matind[2] = 0;          matind[4] = 0;
    matval[0] = -1.0;  matval[2] = 1.0;        matval[4] = 1.0;

    matind[1] = 1;     matind[3] = 1;          matind[5] = 1;
    matval[1] = 1.0;   matval[3] = -3.0;       matval[5] = 1.0;

        lb[0] = 35.0;      lb[1] = 0.0;           lb[2]  = 0.0;
        ub[0] = 40.0;      ub[1] = CPX_INFBOUND;  ub[2]  = CPX_INFBOUND;

   colname[0] = “x1”; colname[1] = “x2”;      colname[2] = “x3”;

   status = CPXaddcols (env, lp, NUMCOLS, NUMNZ, obj, matbeg, matind,
                        matval, lb, ub, colname);
   if ( status )  goto TERMINATE;

TERMINATE:

   return (status);

}  /* END populatebycolumn */


/* The callback function will print out the Phase of the simplex method,
   the sum of infeasibilities if in Phase 1, or the objective if in Phase 2.
   If any of our requests fails, we’ll return an indication to abort.
 */

#ifndef  CPX_PROTOTYPE_MIN
static int CPXPUBLIC
mycallback (CPXENVptr env, void *cbdata, int wherefrom, void *cbhandle)
#else
static int CPXPUBLIC
mycallback (env, cbdata, wherefrom, cbhandle)
CPXENVptr  env;
void       *cbdata;
int        wherefrom;
void       *cbhandle;
#endif
{
   int    status = 0;
   int    phase  = -1;
   double suminf_or_objective;
   int    itcnt = -1;

   if ( wherefrom == CPX_CALLBACK_PRIMAL ) {
      status = CPXgetcallbackinfo (env, cbdata, wherefrom,
                                   CPX_CALLBACK_INFO_ITCOUNT, &itcnt);
      if ( status )  goto TERMINATE;

      status = CPXgetcallbackinfo (env, cbdata, wherefrom,
                                   CPX_CALLBACK_INFO_PRIMAL_FEAS, &phase);
      if ( status )  goto TERMINATE;

      if ( phase == 0 ) {
         status = CPXgetcallbackinfo (env, cbdata, wherefrom,
                                      CPX_CALLBACK_INFO_PRIMAL_INFMEAS,
                                      &suminf_or_objective);
         if ( status )  goto TERMINATE;

         printf (“Iteration %d: Infeasibility measure = %f\n”,
                  itcnt, suminf_or_objective);
      }
      else {
         status = CPXgetcallbackinfo (env, cbdata, wherefrom,
                                      CPX_CALLBACK_INFO_PRIMAL_OBJ,
                                      &suminf_or_objective);
         if ( status )  goto TERMINATE;

         printf (“Iteration %d: Objective = %f\n”,
                  itcnt, suminf_or_objective);
      }

   }

TERMINATE:

   return (status);

} /* END mycallback */
```

Control Callbacks for IloCplex

Control callbacks allow you to control the branch & cut search during the optimization of MIP problems. The following control callbacks are available for `IloCplex`:

- ◆ The node `callback` allows you to query and optionally overwrite the next node CPLEX will process during a branch & cut search.
- ◆ The solve `callback` allows you to specify and configure the optimizer option to be used for solving the LP at each individual node.

- The cut `callback` allows you to add problem-specific cuts at each node.
- The heuristic `callback` allows you to implement a heuristic that tries to generate a new incumbent from the solution of the LP relaxation at each node.
- The branch `callback` allows you to query and optionally overwrite the way CPLEX will branch at each node.

These callbacks are implemented as an extension of the diagnostic callback class hierarchy. This extension is shown below along with the macro names for each of the control callbacks (see Diagnostic Callbacks on page 294 for a discussion of how macros and callback implementation classes relate).

```
IloCplex::MIPCallbackI                       ILOMIPCALLBACKn
   |
   +--- IloCplex::NodeCallbackI              ILONODECALLBACKn
   |
   +--- IloCplex::ControlCallbackI
           |
           +--- IloCplex::BranchCallbackI    ILOBRANCHCALLBACKn
           |
           +--- IloCplex::CutCallbackI       ILOCUTCALLBACKn
           |
          +--- IloCplex::HeuristicCallbackI  ILOHEURISTICCALLBACKn
           |
           +--- IloCplex::SolveCallbackI     ILOSOLVECALLBACKn
```

Similar to class `IloCplex::CallbackI`, class `IloCplex::ControlCallbackI` is not provided for deriving user callback classes, but for defining the common interface for its derived classes. This interface provides methods for querying information about the current node, such as current bounds or solution information for the current node. See class `IloCplex::ControlCallbackI` in the *ILOG CPLEX Reference Manual* for more information.

Example: Controlling Cuts

This example shows how to use the cut callback in the context of solving the *noswot model*. This is a relatively small model from the MIPLIB 3.0 test-set, consisting only of 128 variables. This model is very hard to solve by itself, in fact until the release of CPLEX 6.5 it appeared to be unsolvable even after days of computation.

While it is now solvable directly, the computation time is in the order of several hours on state-of-the-art computers. However, cuts can be derived, the addition of which make the problem solvable in a matter of minutes or seconds. These cuts are:

```
x21 - x22 <= 0
x22 - x23 <= 0
x23 - x24 <= 0
2.08*x11 + 2.98*x21 + 3.47*x31 + 2.24*x41 + 2.08*x51 +
0.25*w11 + 0.25*w21 + 0.25*w31 + 0.25*w41 + 0.25*w51 <= 20.25
```

```
   2.08*x12 + 2.98*x22 + 3.47*x32 + 2.24*x42 + 2.08*x52 +
   0.25*w12 + 0.25*w22 + 0.25*w32 + 0.25*w42 + 0.25*w52 <= 20.25
   2.08*x13 + 2.98*x23 + 3.47*x33 + 2.24*x43 + 2.08*x53 +
   0.25*w13 + 0.25*w23 + 0.25*w33 + 0.25*w43 + 0.25*w53 <= 20.25
   2.08*x14 + 2.98*x24 + 3.47*x34 + 2.24*x44 + 2.08*x54 +
   0.25*w14 + 0.25*w24 + 0.25*w34 + 0.25*w44 + 0.25*w54 <= 20.25
   2.08*x15 + 2.98*x25 + 3.47*x35 + 2.24*x45 + 2.08*x55 +
   0.25*w15 + 0.25*w25 + 0.25*w35 + 0.25*w45 + 0.25*w55 <= 16.25
```

These cuts have been derived after interpreting the model as a resource allocation model on five machines with scheduling, horizon constraints and transaction times. The first tree cuts break symmetries among the machines, while the others capture minimum bounds on transaction costs. See “*MIP: Theory and Practice —Closing the Gap*” for more on how these cuts have been found.

Of course the best way to solve the *noswot* model with these cuts is to simply add the cuts to the model before calling the optimizer. However, for demonstration purposes, we will add the cuts, using a cut callback, only when they are violated at a node. This cut callback takes a list of cuts as parameter and adds individual cuts whenever they are violated with the current LP solution. Notice, that adding cuts does not change the extracted model, but affects only the internal problem representation of the CPLEX object.

This callback is implemented with the code:

```
ILOCUTCALLBACK3(CtCallback, IloExprArray, lhs, IloNumArray, rhs, IloNum, eps) {
  IloInt n = lhs.getSize();
  for (IloInt i = 0; i < n; ++i) {
    IloNum xrhs = rhs[i];
    if ( xrhs < IloInfinity && getValue(lhs[i]) > xrhs + eps ) {
      IloRange cut;
      try {
        cut = (lhs[i] <= xrhs);
        add(cut).end();
        rhs[i] = IloInfinity;
      }
      catch (...) {
        cut.end();
        throw;
      }
    }
  }
}
```

This defines the class `CtCallbackI` as a derived class of `IloCplex::CutCallbackI` and provides the implementation for its virtual methods `main()` and `makeClone()`. It also implements a function `CtCallback` that creates an instance of `CtCallbackI` and returns an `IloCplex::Callback` handle for it.

As indicated by the `3` in the macro name, the constructor of `IloCtCallbackI` takes three parameters, called `lhs`, `rhs`, and `eps`. The constructor stores them as private members to have direct access to them in the callback function, implemented as method `main`. Notice the comma (,) between the type and the argument object in the macro invocation. Here is how the macro expands:

```
class IloCtCallbackI : public IloCplex::FractionalCutCallbackI {
  IloExprArray lhs;
  IloNumArray  rhs;
  IloNum       eps;
public:
  IloCplex::CallbackI* makeClone() const {
    return (new (getEnv()) IloCtCallbackI(*this));
  }
  IloCtCallbackI(IloExprArray xlhs, IloNumArray xrhs, IloNum xeps)
    : lhs(xlhs), rhs(xrhs), eps(xeps)
  {}
  void main();
};

IloCplex::Callback IloCtCallback(IloEnv env,
                                 IloExprArray lhs,
                                 IloNumArray rhs,
                                 IloNum eps) {
  return (IloCplex::Callback(new (env) IloCtCallbackI(lhs, rhs, eps)));
}

void IloCtCallbackI::main() {
  ...
}
```

where the actual implementation code has been substituted with "`...`". Similar macros are provided for other numbers of parameters ranging from 0 through 7 for all callback classes.

The first parameter `lhs` is an array of expressions, and the parameter `rhs` is an array of values. These parameters are the left-hand side and right-hand side values of cuts of the form `lhs <= rhs` to be tested for violation and potentially added. The third parameter `eps` gives a tolerance by which a cut must at least be violated in order to be added to the problem being solved.

The implementation of this example cut callback looks for cuts that are violated by the current LP solution of the node where the callback is invoked. We loop over the potential cuts, checking each for violation by querying the value of the `lhs` expression with respect to the current solution. This is done by calling `getValue` with this expression as a parameter. This is tested for violation of more than the tolerance parameter `eps` with the corresponding right-hand side value.

Tip: A numerical tolerance is always a wise thing to consider when dealing with any non-trivial model, to avoid certain logical inconsistencies that could otherwise occur due to numerical roundoff. Here we use the standard CPLEX simplex feasibility tolerance for this purpose, to insure some consistency with the way CPLEX is treating the rest of the model.

If a violation is detected, the callback creates an `IloRange` object to represent the cut: `lhs[i] <= rhs[i]`. It is added to the LP by calling method `add()`. Adding a cut to CPLEX, unlike extracting a model, only copies the cut into the CPLEX data structures, without maintaining a notification link between the two. Thus, after a cut has been added, it can be deleted by calling its method `end()`. In fact, it should be deleted, as otherwise the memory used for the cut could not be reclaimed. For convenience, method `add()` returns the cut that has been added, and thus we can call `end()` directly on the returned `IloRange` object.

It is important that all resources that have been allocated during a callback are freed again before leaving the callback--even in the case of an exception. Here exceptions could be thrown when creating the cut itself or when trying to add it, for example, due to memory exhaustion. Thus, we enclose these operations in a try block and catch all exceptions that may occur. In the case of an exception, we delete the cut by calling `cut.end()` and re-throw whatever exception was caught. Re-throwing the exception can be omitted if you want to continue the optimization without the cut.

After the cut has been added, we set the `rhs` value to `IloInfinity` to avoid checking this cut for violation at the next invocation of the callback. Note that we did not simply remove the *i*th element of arrays `rhs` and `lhs`, because this is not supported if the cut callback is invoked from a parallel optimizer. However, changing array elements is allowed.

Also, for the potential use of the callback in parallel, the variable `xrhs` ensures that we are using the same value when checking for violation of the cut as when adding the cut. Otherwise, another thread may have set the `rhs` value to `IloInfinity` just between the two actions, and a useless cut would be added. CPLEX would actually handle this correctly, as it handles adding the same cut from different threads.

Function `makeCuts()` generates the arrays `rhs` and `lhs` to be passed to the cut callback. It first declares the array of variables to be used for defining the cuts. Since the environment is not passed to the constructor of that array, an array of 0-variable handles is created. In the following loop, these variable handles are initialized to the correct variables in the `noswot` model which are passed to this function as parameter `vars`. The identification of the variables is done by querying variables names. Once all the variables have been assigned, they are used to create the `lhs` expressions and `rhs` values of the cuts.

The cut callback is created and passed to CPLEX in the line:

```
cplex.use(CtCallback(env, lhs, rhs, cplex.getParam(IloCplex::EpRHS)));
```

The function `CtCallback` constructs an instance of our callback class `CtCallbackI` and returns an `IloCplex::Callback` handle object for it. This is directly passed to function `cplex.use`.

We should point out that `IloCplex` provides an easier way to manage such cuts in a case like this, where all cuts can be easily enumerated before starting the optimization. Calling the methods `cplex.addCut()` and `cplex.addCuts()` allows you to copy the cuts to

`IloCplex` before the optimization. Thus, instead of creating and using the callback, we could have written:

```
cplex.addCuts(makeCuts(var));
```

as shown in example `iloadmipex7.cpp` in the distribution. During branch & cut, CPLEX will consider adding individual cuts to its representation of the model only if they are violated by a node LP solution in about the same way this example handles them. Whether this or adding the cuts directly to the model gives better performance when solving the model depends on the individual problem.

Complete Program: iloadmipex5.cpp

The complete program, `iloadmipex5.cpp`, appears here or online in the standard distribution.

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

ILOCUTCALLBACK3(CtCallback, IloExprArray, lhs, IloNumArray, rhs, IloNum, eps) {
  IloInt n = lhs.getSize();
  for (IloInt i = 0; i < n; ++i) {
    IloNum xrhs = rhs[i];
    if ( xrhs < IloInfinity  &&  getValue(lhs[i]) > xrhs + eps ) {
      IloRange cut;
      try {
        cut = (lhs[i] <= xrhs);
        add(cut).end();
        rhs[i] = IloInfinity;
      }
      catch (...) {
        cut.end();
        throw;
      }
    }
  }
}

void
makeCuts(const IloNumVarArray vars, IloExprArray lhs, IloNumArray rhs) {
  IloNumVar x11, x12, x13, x14, x15;
  IloNumVar x21, x22, x23, x24, x25;
  IloNumVar x31, x32, x33, x34, x35;
  IloNumVar x41, x42, x43, x44, x45;
  IloNumVar x51, x52, x53, x54, x55;
  IloNumVar w11, w12, w13, w14, w15;
  IloNumVar w21, w22, w23, w24, w25;
  IloNumVar w31, w32, w33, w34, w35;
  IloNumVar w41, w42, w43, w44, w45;
  IloNumVar w51, w52, w53, w54, w55;
  IloInt num = vars.getSize();

  for (IloInt i = 0; i < num; ++i) {
     if      ( strcmp(vars[i].getName(), "X11") == 0 ) x11 = vars[i];
     else if ( strcmp(vars[i].getName(), "X12") == 0 ) x12 = vars[i];
     else if ( strcmp(vars[i].getName(), "X13") == 0 ) x13 = vars[i];
     else if ( strcmp(vars[i].getName(), "X14") == 0 ) x14 = vars[i];
     else if ( strcmp(vars[i].getName(), "X15") == 0 ) x15 = vars[i];
     else if ( strcmp(vars[i].getName(), "X21") == 0 ) x21 = vars[i];
     else if ( strcmp(vars[i].getName(), "X22") == 0 ) x22 = vars[i];
     else if ( strcmp(vars[i].getName(), "X23") == 0 ) x23 = vars[i];
     else if ( strcmp(vars[i].getName(), "X24") == 0 ) x24 = vars[i];
     else if ( strcmp(vars[i].getName(), "X25") == 0 ) x25 = vars[i];
     else if ( strcmp(vars[i].getName(), "X31") == 0 ) x31 = vars[i];
     else if ( strcmp(vars[i].getName(), "X32") == 0 ) x32 = vars[i];
     else if ( strcmp(vars[i].getName(), "X33") == 0 ) x33 = vars[i];
     else if ( strcmp(vars[i].getName(), "X34") == 0 ) x34 = vars[i];
     else if ( strcmp(vars[i].getName(), "X35") == 0 ) x35 = vars[i];
     else if ( strcmp(vars[i].getName(), "X41") == 0 ) x41 = vars[i];
     else if ( strcmp(vars[i].getName(), "X42") == 0 ) x42 = vars[i];
     else if ( strcmp(vars[i].getName(), "X43") == 0 ) x43 = vars[i];
     else if ( strcmp(vars[i].getName(), "X44") == 0 ) x44 = vars[i];
     else if ( strcmp(vars[i].getName(), "X45") == 0 ) x45 = vars[i];
     else if ( strcmp(vars[i].getName(), "X51") == 0 ) x51 = vars[i];
     else if ( strcmp(vars[i].getName(), "X52") == 0 ) x52 = vars[i];
     else if ( strcmp(vars[i].getName(), "X53") == 0 ) x53 = vars[i];
     else if ( strcmp(vars[i].getName(), "X54") == 0 ) x54 = vars[i];
     else if ( strcmp(vars[i].getName(), "X55") == 0 ) x55 = vars[i];
     else if ( strcmp(vars[i].getName(), "W11") == 0 ) w11 = vars[i];
     else if ( strcmp(vars[i].getName(), "W12") == 0 ) w12 = vars[i];
     else if ( strcmp(vars[i].getName(), "W13") == 0 ) w13 = vars[i];
     else if ( strcmp(vars[i].getName(), "W14") == 0 ) w14 = vars[i];
     else if ( strcmp(vars[i].getName(), "W15") == 0 ) w15 = vars[i];
     else if ( strcmp(vars[i].getName(), "W21") == 0 ) w21 = vars[i];
     else if ( strcmp(vars[i].getName(), "W22") == 0 ) w22 = vars[i];
     else if ( strcmp(vars[i].getName(), "W23") == 0 ) w23 = vars[i];
     else if ( strcmp(vars[i].getName(), "W24") == 0 ) w24 = vars[i];
     else if ( strcmp(vars[i].getName(), "W25") == 0 ) w25 = vars[i];
     else if ( strcmp(vars[i].getName(), "W31") == 0 ) w31 = vars[i];
     else if ( strcmp(vars[i].getName(), "W32") == 0 ) w32 = vars[i];
     else if ( strcmp(vars[i].getName(), "W33") == 0 ) w33 = vars[i];
     else if ( strcmp(vars[i].getName(), "W34") == 0 ) w34 = vars[i];
     else if ( strcmp(vars[i].getName(), "W35") == 0 ) w35 = vars[i];
     else if ( strcmp(vars[i].getName(), "W41") == 0 ) w41 = vars[i];
     else if ( strcmp(vars[i].getName(), "W42") == 0 ) w42 = vars[i];
     else if ( strcmp(vars[i].getName(), "W43") == 0 ) w43 = vars[i];
     else if ( strcmp(vars[i].getName(), "W44") == 0 ) w44 = vars[i];
     else if ( strcmp(vars[i].getName(), "W45") == 0 ) w45 = vars[i];
     else if ( strcmp(vars[i].getName(), "W51") == 0 ) w51 = vars[i];
     else if ( strcmp(vars[i].getName(), "W52") == 0 ) w52 = vars[i];
     else if ( strcmp(vars[i].getName(), "W53") == 0 ) w53 = vars[i];
     else if ( strcmp(vars[i].getName(), "W54") == 0 ) w54 = vars[i];
     else if ( strcmp(vars[i].getName(), "W55") == 0 ) w55 = vars[i];
  }
  lhs.add(x21 - x22);  rhs.add(0.0);
```

```
  lhs.add(x22 - x23);  rhs.add(0.0);
  lhs.add(x23 - x24);  rhs.add(0.0);
  lhs.add(2.08*x11 + 2.98*x21 + 3.47*x31 + 2.24*x41 + 2.08*x51 +
          0.25*w11 + 0.25*w21 + 0.25*w31 + 0.25*w41 + 0.25*w51);
rhs.add(20.25);
  lhs.add(2.08*x12 + 2.98*x22 + 3.47*x32 + 2.24*x42 + 2.08*x52 +
          0.25*w12 + 0.25*w22 + 0.25*w32 + 0.25*w42 + 0.25*w52);
rhs.add(20.25);
  lhs.add(2.08*x13 + 2.98*x23 + 3.47*x33 + 2.24*x43 + 2.08*x53 +
          0.25*w13 + 0.25*w23 + 0.25*w33 + 0.25*w43 + 0.25*w53);
rhs.add(20.25);
  lhs.add(2.08*x14 + 2.98*x24 + 3.47*x34 + 2.24*x44 + 2.08*x54 +
          0.25*w14 + 0.25*w24 + 0.25*w34 + 0.25*w44 + 0.25*w54);
rhs.add(20.25);
  lhs.add(2.08*x15 + 2.98*x25 + 3.47*x35 + 2.24*x45 + 2.08*x55 +
          0.25*w15 + 0.25*w25 + 0.25*w35 + 0.25*w45 + 0.25*w55);
rhs.add(16.25);
}


int
main(int argc, char** argv)
{
  IloEnv env;
  try {
    IloModel m;
    IloCplex cplex(env);

    IloObjective   obj;
    IloNumVarArray var(env);
    IloRangeArray  con(env);

    env.out() << "reading ../../../examples/data/noswot.mps" << endl;
    cplex.importModel(m, "../../../examples/data/noswot.mps", obj, var, con);

    env.out() << "constructing cut callback ..." << endl;

    IloExprArray lhs(env);
    IloNumArray  rhs(env);
    makeCuts(var, lhs, rhs);
    cplex.use(CtCallback(env, lhs, rhs, cplex.getParam(IloCplex::EpRHS)));

    env.out() << "extracting model ..." << endl;
    cplex.extract(m);

    env.out() << "solving model ...\n";
    cplex.solve();
    env.out() << "solution status is " << cplex.getStatus() << endl;
    env.out() << "solution value  is " << cplex.getObjValue() << endl;
  }
  catch (IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
```

```
}
```

Using Parallel Optimizers

This section tells you how to use ILOG CPLEX parallel optimizers. It includes sections on:

- Parallel Libraries
- Threads
- Nondeterminism
- Clock Settings and Time Measurement
- Using Parallel Optimizers in the Interactive Optimizer
- Using Parallel Optimizers in the CPLEX Component Libraries
- Parallel MIP Optimizer
- Parallel Barrier Optimizer
- Parallel Simplex Optimizer

There are three specialized ILOG CPLEX optimizers—Parallel Simplex, Parallel MIP, and Parallel Barrier—implemented to run on hardware platforms with parallel processors. These parallel optimizers, like other ILOG CPLEX optimizers, are available in the Interactive Optimizer and in the Component Libraries, if you hold a ILOG CPLEX Parallel license. The parallel license allows you to use the parallel implementation of the ILOG CPLEX optimizers for which you already hold a license. For example, if you are licensed to use the ILOG CPLEX Interactive Optimizer, the MIP Optimizer, and the parallel optimizers, then Parallel Simplex and Parallel MIP will both be available to you (if they have been implemented on your parallel platform). If you then add a license for the ILOG CPLEX Barrier Optimizer, the Parallel Barrier Optimizer will automatically be available to you as well.

For Windows users, or for LINUX users, no special procedures are needed to compile and link your program to the parallel libraries. For other UNIX platforms, separate parallel versions of the libraries and Interactive Optimizer are provided for your use. Table 8.8 summarizes these details. Additional compiler/linker flags may be needed when compiling your program to use parallel CPLEX. Consult the makefile that is provided in the CPLEX distribution for your computer platform, and if there is a line marked "For parallel" use the information there as a guide.

Parallel Libraries

Generally, you use ILOG CPLEX parallel optimizers just as you use ILOG CPLEX serial optimizers. They are available through the Interactive Optimizer and through methods and routines of the Callable Library.

To access the parallel optimizers from routines of the Component Libraries, you need to link to the parallel library. Table 8.8 summarize the names of the serial and parallel libraries for UNIX platforms.

Table 8.8 *ILOG CPLEX Serial and Parallel Libraries for UNIX Platforms*

Most Unix platforms	Serial Library	Parallel Library
Interactive Optimizer	`cplex`	`parcplex`
Component Libraries	`libcplex.a`	`libparcplex.a`

Threads

The ILOG CPLEX parallel optimizers are licensed for a specific maximum number of *threads* (that is, the number processors applied to a given problem). The number of threads that ILOG CPLEX actually uses during a parallel optimization is the *smaller* of:

- the number of threads made available by the operating system;
- the number of threads indicated by the *licensed* values of the thread-limit parameters. Table 8.9 summarizes the values of those thread-limit parameters.

Table 8.9 *Thread-Limit Parameters*

Interactive Commands	Concert Technology Enumeration Value	Callable Library Parameter
`set simplex limits threads`	`IloCplex::SimThreads`	`CPX_PARAM_SIMTHREADS`
`set barrier limits threads`	`IloCplex::BarThreads`	`CPX_PARAM_BARTHREADS`
`set mip limits threads`	`IloCplex::MIPThreads`	`CPX_PARAM_MIPTHREADS`
`set mip limits strongthreads`	`IloCplex::StrongThreadLim`	`CPX_PARAM_STRONGTHREADLIM`

The notion of the number of threads used when running a parallel CPLEX optimizer is entirely separate from the limit on licensed uses. A typical CPLEX license permits one licensed use, that is a single concurrent execution on one licensed computer. If the license also contains the parallel option with a thread limit of, say, four (on a machine with at least four processors), that one concurrent execution of CPLEX can employ any number of parallel threads to increase performance, up to that limit of 4. A license with the parallel option, that additionally has a limit larger than one on the number of licensed uses, can support that many simultaneous executions of CPLEX, each with the licensed maximum number of parallel threads. In such a case, the operating system will manage any contention for processors.

The number of parallel threads used by a CPLEX optimizer is controlled by operating system environment variables and/or CPLEX parameter settings, up to the limits found in the CPLEX license on the machine. This is discussed in more detail in the sections that follow.

Threads and Platform Considerations

Ordinarily, your operating system will make available to ILOG CPLEX as many threads as there are processors on your machine. In some cases, you can override this behavior through operating-system environment variables.

◆ DEC only: The default number of threads is the number of processors on the machine. You can override this default by setting the operating system environment variable `MP_THREAD_COUNT` before you call ILOG CPLEX.

◆ SGI only: The default number of threads is the smaller of 8 or the number of processors on the machine. You can override this default by setting the operating system environment variable `MPC_NUM_THREADS` before you call ILOG CPLEX.

Those environment variables can be used to establish an upper limit on thread count, subject to the limit of your ILOG CPLEX parallel license.

Individual ILOG CPLEX optimizers, such as the ILOG CPLEX Barrier Optimizer, may be affected by other platform considerations. See the various parallel optimizers (MIP on page 325, nested on page 328, barrier on page 329, or simplex on page 330) for details that cover those considerations.

Example: Threads and Licensing

For example, let's assume you use ILOG CPLEX to optimize MIP models on an eight-processor machine, and you have purchased a ILOG CPLEX license for four parallel threads. Then you can use the Interactive Optimizer command `set mip limit threads i`, substituting values 1 through 4 for `i`. Even if you set an operating system environment variable to `6`, you will not be able to set `mip limit threads` higher than `4` because you are licensed for a maximum of four threads. In contrast, if you set an operating system environment variable to `2`, then you can set `mip limit threads` only as large as `2`, and any MIP optimization you carry out will be limited to two processors because of the setting of the operating system environment variable.

Threads and Performance Considerations

If you set the number of threads to a value greater than the number of processors, performance will usually degrade. If you set the number of threads to a value less than the number of processors, the remaining processors will be available for other jobs on your platform. Simultaneously running multiple parallel jobs with a total number of threads exceeding the number of processors may impair the performance of each individual process as its threads compete with one another.

If you set an operating system environment variable to a greater value than you actually use within ILOG CPLEX with a `limit threads` parameter, your operating system may create idle threads that still consume system resources. If you know in advance how many threads you want to use, we recommend that you set the operating system environment variable to that number before you start ILOG CPLEX.

The benefit of applying more threads to optimizing a specific problem varies depending on the optimizer you use and the characteristics of the problem. You should experiment to assess performance improvements and degradation when you apply more or fewer processors. For example, when you optimize an LP relaxation, there may be little or no benefit in applying more than four processors to the task. In contrast, if you use 16 processors during the MIP phase of an optimization, you may improve solution speed by a factor of 20. In such a case, you should set the parameters `simplex limit threads` and `mip limit threads` to *different* values in order to use your computing resources efficiently.

Another key consideration in setting optimizer and global thread limits is your management of overall system load.

Nondeterminism

Among the ILOG CPLEX parallel optimizers, only parallel simplex follows a deterministic algorithm, producing the same number of iterations and the same solution path when you apply it to the same problem more than once. In contrast, the parallel barrier and parallel MIP optimizers are *nondeterministic*: repeated solutions of a model using exactly the same settings can produce different solution paths and, in the case of the parallel MIP optimizer, very different solution times and results.

The basic algorithm in the ILOG CPLEX Parallel MIP Optimizer is branch & cut. The primary source of parallelism in branch & cut is the solution of the LP subproblems at the individual nodes of the search tree. These subproblems can be distributed over available processors to be carried out in parallel. The individual solution paths for these subproblems will, in fact, be deterministic, but the speed at which their solutions occur can vary slightly. These variations lead to nodes being taken from and replaced in the branch & cut tree in different order, and this reordering leads to nondeterminism about many other quantities that control the optimization. This nondeterminism is unavoidable in such a context, and its effects can result in some cases in very different solution paths.

Clock Settings and Time Measurement

The clock-type parameter determines how ILOG CPLEX measures computation time. For most nonparallel processing purposes, CPU time, the default type, is appropriate. It reports how much time the CPU was actually employed to complete an operation. This value is highly system dependent. On some parallel systems, it may measure aggregate CPU time, that is, the sum of time used by all processors, or on others, it may report the CPU time of only one process. In short, it may give you a misleading indication of parallel speed.

The alternative type, wall-clock time, is usually more appropriate for parallel computing because it measures the total physical time elapsed after an operation begins. When multiple processes are active, and when parallel optimizers are active, wall-clock time can be much different from CPU time.

You can choose the type of clock setting, in the:

- Interactive Optimizer, with the command `set clocktype` *`i`*.
- Concert Technology Library, use the method `IloCplex::setParam(ClockType,` *`i`*`)`.
- Callable Library, use the routine `CPXsetintparam(env, CPX_PARAM_CLOCKTYPE,` *`i`*`)`.

Replace the *`i`* with the value `1` to specify CPU time or `2` to specify wall-clock time.

Using Parallel Optimizers in the Interactive Optimizer

1. If necessary for your platform, set any operating system environment variable needed for parallel operation. See *Threads* on page 320 and platform considerations for the various parallel optimizers (MIP on page 325, nested on page 328, barrier on page 329, or simplex on page 330) for details.
2. Start the parallel implementation of the ILOG CPLEX Interactive Optimizer with the command `parcplex` (or `cplex` on machines where a separate executable is not needed) at the operating system prompt.
3. Set the thread-limit, as explained in *Threads* on page 320.
4. Enter and populate your problem object as usual.
5. Call the parallel optimizer with the appropriate command:

Parallel MIP Optimizer	`mipopt`
Parallel Barrier Optimizer	`baropt`
Parallel Simplex Optimizer	`primopt` or `tranopt`

Using Parallel Optimizers in the CPLEX Component Libraries

1. Link your application to the parallel implementation of the ILOG CPLEX Component Libraries. See Table 8.8 on page 320 for the library name, which varies according to platform.
2. Create your ILOG CPLEX environment and initialize a problem object in the usual way.

See *Initialize the ILOG CPLEX Environment* on page 57 and *Instantiate the Problem Object* on page 58 for details.

3. Within your application, set the appropriate CPLEX parameter from Table 8.9 to specify the number of threads.

4. Enter and populate your problem object in the usual way, as in *Put Data in the Problem Object* on page 58.

5. Call the parallel optimizer with the appropriate method or routine:

Optimizer	**Concert Technology Library**	**Callable Library**
Parallel MIP Optimizer	`IloCplex::solve()`	`CPXmipopt()`
Parallel Barrier Optimizer	`IloCplex::Barrier` or `IloCplex::BarrierPrimal` or `IloCplex::BarrierDual`	`CPXbaropt()` or `CPXhybbaropt()`
Parallel Simplex Optimizer	`IloCplex::Primal` or `IloCplex::Dual`	`CPXprimopt()` or `CPXdualopt()`

Parallel MIP Optimizer

The CPLEX Parallel MIP Optimizer is quite robust with respect to parallelism, so it achieves remarkable speedups on a wide variety of models—particularly difficult ones that process a large number of nodes in the branch & cut search tree while proving optimality. The parallel MIP optimizer provides several different opportunities for applying multiple processors to the solution of a problem.

- Parallelism can be applied to the *root relaxation* using either parallel barrier optimizer or (on platforms where it is available) parallel simplex optimizer, depending on the setting of the start-algorithm parameter. Parallelism here is controlled by the barrier (or simplex) thread-limit parameter.
- Once the root relaxation has been solved, you can process nodes in the branch & cut tree in parallel by setting the MIP thread-limit parameter to a value greater than `1`.
  - In the Interactive Optimizer, use the command `set mip limits threads`.
  - When using the Component Libraries, set the parameter `IloCplex::MipThreads` or `CPX_PARAM_MIPTHREADS`.
- Alternatively, you can process one node at a time but apply multiple processors to the solution of each node by setting the MIP thread-limit to `1` (one) and choosing either parallel simplex or parallel barrier for the subalgorithm.

- A third alternative, using strong branching as the variable selection strategy, is to process one node at a time but apply multiple processors to strong branching variable selection, by setting the MIP thread-limit to `1` (one) and the strong branching thread-limit to a value greater than one.

The following sections discuss details and tradeoffs associated with these options.

Platform Considerations

The parallel MIP optimizer is available on all the parallel platforms that ILOG CPLEX supports.

Memory Considerations and the Parallel MIP Optimizer

Before the parallel MIP optimizer invokes parallel processing, it makes separate, internal copies of the initial problem. The individual processors use these copies during computation, so each of them requires an amount of memory roughly equal to the original model after it is presolved.

Output from the Parallel MIP Optimizer

The parallel MIP optimizer generates slightly different output from the serial MIP optimizer (described in *Termination* on page 166 and *Post-Solution Information in a MIP* on page 167). The following paragraphs explain those differences.

Timing Statistics from the Parallel MIP Optimizer

We explained that you can control the amount of information that ILOG CPLEX displays and records in its log files.

To make ILOG CPLEX record elapsed time for the MIP optimizer:

- In the Interactive Optimizer, use the command `set mip display i`, where `i` is `1`, `2`, `3`, `4`, or `5`.
- When using the CPLEX Component Libraries, set the parameter `IloCplex::MIPDisplay` or `CPX_PARAM_MIPDISPLAY` to one of these same values `1 - 5`.

In the parallel MIP optimizer, these elapsed times are always wall-clock times, regardless of the clock-type parameter.

ILOG CPLEX prints a summary of timing statistics specific to the parallel MIP optimizer at the end of optimization. You can see typical timing statistics in the following sample run.

```
Problem 'fixnet6.mps' read.
Read time =    0.04 sec.
CPLEX> o
Tried aggregator 1 time.
MIP Presolve modified 308 coefficients.
Aggregator did 1 substitutions.
Reduced MIP has 477 rows, 877 columns, and 1754 nonzeros.
Presolve time =    0.02 sec.
Clique table members: 2
MIP emphasis: optimality
Root relaxation solution time =    0.04 sec.

        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer     Best Node    ItCnt     Gap

      0     0     3192.0420    12                   3192.0420      305
                  3263.9220    19                   Cuts:  36      341
                  3393.0917    17                   Cuts:  24      403
                  3444.9996    19                Flowcuts:  9      439
                  3479.7206    24                Flowcuts:  6      470
                  3489.7893    21                Flowcuts:  3      482
                  3500.4789    24                Flowcuts:  4      494
                  3502.0646    26                Flowcuts:  4      499
                  3526.8260    20                Flowcuts:  2      502
                  3527.0669    19                Flowcuts:  1      504
                  3527.2559    22                Flowcuts:  1      506
                  3527.6402    24                Flowcuts:  1      508
                  3529.7853    18                Flowcuts:  1      515
*     0+    0     4116.0000     0     4116.0000     3529.7853      515   14.24%
*    40+   24     4077.0000     0     4077.0000     3808.7017     1191    6.58%
*    46    23     3983.0000     0     3983.0000     3854.2312     1198    3.23%


Sequential (before b&b):
  CPU      time            =    0.79
Parallel b&b, 4 threads:
  Real     time            =    0.48
  Critical time (total)    =    0.00
  Spin     time (average)  =    0.00
                              -------
Total (sequential+parallel) =    1.27 sec.

Cover cuts applied:  1
Flow cuts applied:  43
Gomory fractional cuts applied: 8

Integer optimal solution:  Objective =    3.9830000000e+03
Solution time =    1.65 sec.  Iterations = 1415  Nodes = 74
```

The summary at the end of the sample tells us that *0.79* seconds were spent in CPU time (since the clock-type parameter was the default) in the sequential phase, mainly in preprocessing by the presolver and in solving the initial linear-programming relaxation. The parallel part of this sample run took *0.48* seconds of real time (that is, elapsed time for that phase).

Other parts of the sample report indicate that the processors spent an average of *0.00* seconds of real time spinning (that is, waiting for work while there were too few active nodes available). The real critical time was a total of *0.00* seconds, time spent by individual processors in updating global, shared information. Since only one processor can access the critical region at any instant in time, the amount of time spent in this region really is crucial: any other processor that tries to access this region must wait, thus sitting idle, and this idle time is counted separately from the spin time.

Logs from the Parallel MIP Optimizer

There is also a difference in the way logging occurs in the parallel MIP optimizer. When this optimizer is called, it makes a number of copies of the problem. These copies are known as *clones*. The parallel MIP optimizer creates as many clones as there are threads available to it. When the optimizer exits, these clones and all their paraphernalia are discarded.

If a log file is active when the clones are created, then ILOG CPLEX creates a clone log file for each clone. The clone log files are named `cloneK.log`, where `K` is the index of the clone, ranging from 0 (zero) to the number of threads minus one. Since the clones are created at each call to the parallel MIP optimizer and discarded when it exits, the clone logs are opened at each call and closed at each exit. (The clone log files are not removed when the clones themselves are discarded.)

The clone logs contain information normally recorded in the ordinary log file (by default, `cplex.log`) but inconvenient to send through the normal log channel. The information likely to be of most interest to you are special messages, such as error messages, that result from calls to the LP optimizers called for the subproblems.

Nested Parallelism

If a ILOG CPLEX parallel LP optimizer (for example, parallel simplex or parallel barrier) is available along with the parallel MIP optimizer, then you have the option to choose the strategy for solving the MIP model in parallel in different ways:

◆ Make the branch & cut parallel.

  If you want to make the *branching* parallel, then you should set the thread-limit parameter for the *subproblem* optimizer to 1 (one). For example:

  - In the Interactive Optimizer, use the command `set barrier limit threads 1` or `set simplex limit threads 1`.
  - When using the CPLEX Component Libraries, set the parameters `IloCplex::BarThreads` / `CPX_PARAM_BARTHREADS` or `IloCplex::SimThreads` / `CPX_PARAM_SIMTHREADS` to `1` (one).

◆ Make the solution of subproblems parallel.

  If you want all the parallelism to occur in solutions of the subproblems, not at the branching level, then you should set the MIP thread-limit parameter to `1` (one). For example:

  - In the Interactive Optimizer use the command `set mip limit threads 1`.

- When using the CPLEX Component Libraries, set the parameter `IloCplex::MipThreads` or `CPX_PARAM_MIPTHREADS` to `1` (one).

◆ make the strong branching variable selection parallel;

If you want the strong branching variable selection to occur in parallel, you should set the MIP thread-limit parameter to `1` (one) and the strong branching thread-limit parameter to a value greater than one. For example:

- In the Interactive Optimizer, use the commands `set mip limits threads 1` and `set mip limits strongthreads 2`.
- When using the CPLEX Component Libraries, set the parameter `IloCplex::MipThreads` / `CPX_PARAM_MIPTHREADS` to `1` (one) and to set the parameter `IloCplex::StrongThreadLim` / `CPX_PARAM_STRONGTHREADLIM` to a value greater than one.

◆ make both branch & bound *and* subproblem solutions parallel by *nesting*.

On systems that support nested parallelism, you can make both the branch & cut and the subproblem solutions work in parallel by setting the thread-limit parameters for both optimizers (MIP for the branch & cut, barrier or simplex for the subproblems) to values greater than one. For example, on a six-processor system, If the MIP thread-limit were set to `3`, and the barrier thread-limit were set to `2`, then three subproblems could be solved simultaneously, each using two processors.

Nested Parallelism Platform Considerations

Nested parallelism is *not* supported on DEC, HP, nor SGI parallel platforms. Consequently, on these platforms, you cannot call parallel optimizers from within other parallel optimizers. In particular, you cannot call the ILOG CPLEX Parallel Simplex or ILOG CPLEX Parallel Barrier Optimizer from within the ILOG CPLEX Parallel MIP Optimizer.

For example, if you invoke the MIP optimizer and you have set the MIP thread-limit parameter to use more than one thread, when you call the simplex optimizer on MIP subproblems, it will use the serial, sequential algorithm. To use the parallel simplex optimizer or the parallel barrier optimizer on MIP subproblems, you must set the MIP thread-limit to 1 (one). That is, you must make it serial, sequential (not parallel).

MIP First Rule and its Exceptions

Since the parallelism available in the parallel MIP optimizer is normally applicable to a very large class of problems, and since it normally benefits them greatly, it will more often be best to use the parallel MIP optimizer, rather than parallel simplex or parallel barrier optimizers on the subproblems. We call this rule of thumb, “MIP first.”

One exception to this MIP-first rule occurs in problems with an extremely large aspect ratio (that is, the number of columns divided by number of rows) where the memory requirements of the parallel MIP optimizer exceed available resources.

Another exception, again for problems with a large aspect ratio, occurs for any problem that generates relatively few active nodes during a MIP optimization.

Finally, large aspect problems that spend a large fraction of the total solution time searching for the first incumbent may be good candidates for the parallel simplex optimizer (rather than MIP first).

Parallel Barrier Optimizer

The ILOG CPLEX Parallel Barrier Optimizer achieves significant speedups over its serial counterpart (the ILOG CPLEX Barrier Optimizer described in *Solving LP Problems with the Barrier Optimizer* on page 129, and in Chapter 7, *Solving Quadratic Programming Problems*) on a wide variety of classes of problems. Consequently, the parallel barrier optimizer will be the best LP choice on a parallel computer more frequently than on a single-processor. For that reason, you should be careful *not* to apply performance data or experience based on *serial* optimizers when you are choosing which optimizer to use on a parallel platform.

If you decide to use the parallel barrier optimizer on the *subproblems* of a MIP, see also other special considerations about nested parallelism in *Nested Parallelism* on page 327.

Platform Considerations

On Hewlett-Packard (HP) only, the default number of threads used by the ILOG CPLEX Parallel Barrier Optimizer is the number of processors on the computer. You can override this default by resetting the operating system environment variable `MP_NUMBER_OF_THREADS` before you start ILOG CPLEX.

On Sun only, you must set the UNIX environment variable `PARALLEL` to a number. This number will be the overriding maximum for the number of threads, subject to licensing limits, as explained on page 319. The default behavior of the ILOG CPLEX Parallel Barrier Optimizer without this environment variable is to use only one thread.

Parallel Simplex Optimizer

In the ILOG CPLEX implementation of the Parallel Simplex Optimizer, the column-based work occurs in parallel. Consequently, a significant gain in speed may occur in the dual simplex optimization(`tranopt` / `IloCplex::Dual` / `CPXdualopt()`).

Occasionally, in the primal simplex optimizer if the primal gradient parameter is set to steepest-edge pricing and the aspect ratio (that is, the number of columns divided by the number of rows) is relatively large (for example, *10* or more), then good speedups may occur with the parallel optimizer here, too. Larger problems with a somewhat smaller aspect ratio may also benefit from parallel simplex optimization. Since it is difficult to predict where the breakpoint will be, we encourage you to experiment.

If you decide to use the parallel simplex optimizer on the *subproblems* of a MIP, see other special considerations about nested parallelism in *Nested Parallelism* on page 327.

Platform Considerations

The parallel simplex optimizer is available only on DEC and SGI parallel systems.

A P P E N D I X

A

Interactive Optimizer Commands

The following table lists Interactive Optimizer commands, their primary options, and pages in this manual on which usage examples can be found.

Command		Options		Example
`add`				
`baropt`				page 96, page 133
`baropt`	`dualopt`			page 133
`baropt`	`primopt`			page 133
`baropt`	`stop`			page 133
`change`	`bounds`			
`change`	`coefficient`			
`change`	`delete`			
`change`	`name`			
`change`	`objective`			

Command		Options		Example
change	problem	*type*		page 155, page 168, page 242, page 242, page 246
change	rhs			
change	qpterm			page 243
change	sense	max		page 224
change	type			page 155
display	iis			page 116
display	problem	all		
display	problem	binaries		page 154
display	problem	bounds		
display	problem	constraints		page 245
display	problem	generals		page 154
display	problem	histogram		page 132
display	problem	integers		page 154
display	problem	names		
display	problem	qpvariables		page 245
display	problem	semi-continuous		
display	problem	sos		
display	problem	stats		page 93, page 154
display	problem	variable		page 245
display	sensitivity			
display	settings			
display	settings	all		
display	settings	changed		

Command		Options		Example
display	solution	basis		
display	solution	bestbound		
display	solution	dual		
display	solution	kappa		page 109
display	solution	objective		
display	solution	quality		page 113, page 138, page 245
display	solution	reduced		
display	solution	slacks		
display	solution	variables		
enter				
help				
mipopt				page 156
netopt				page 96, page 222
optimize				page 96
primopt				page 96
quit				
read	*filename*	*type*		page 102, page 222, page 132
set	advance	yes		page 102, page 224
set	barrier			page 133
set	barrier	algorithm	*i*	page 137, page 145, page 244
set	barrier	colnonzeros	*i*	page 142, page 146

Command		Options		Example
`set`	`barrier`	`convergetol`	*i*	page 147
`set`	`barrier`	`crossover`	*i*	
`set`	`barrier`	`display`	*i*	page 133, page 135, page 146
`set`	`barrier limits`	`corrections`	*i*	page 145
`set`	`barrier limits`	`growth`	*i*	
`set`	`barrier limits`	`iterations`	`0`	page 132
`set`	`barrier limits`	`objrange`	*i*	page 148
`set`	`barrier limits`	`varupper`	*i*	page 147
`set`	`barrier`	`ordering`	*i*	page 143
`set`	`barrier`	`startalg`	*i*	page 144
`set`	`clocktype`		*i*	
`set`	`defaults`			
`set`	`logfile`	*`filename`*		
`set`	`lpsolver`		*i*	
`set`	`mip cuts`	*`class`*	`-1`	page 160, page 185
`set`	`mip`	`display`	`2`	page 171
`set`	`mip`	`emphasis`		
`set`	`mip`	`interval`	`100`	page 171
`set`	`mip limits`	`aggforcut`	*i*	
`set`	`mip limits`	`cutsfactor`		page 161
`set`	`mip limits`	`cutpasses`	*i*	
`set`	`mip limits`	`gomorycand`	*i*	
`set`	`mip limits`	`gomorypass`	*i*	
`set`	`mip limits`	`nodes`		page 166

Command		Options		Example
set	mip limits	solutions		page 166
set	mip limits	strongcand	*i*	
set	mip limits	strongit	*i*	
set	mip limits	treememory		page 182, page 183
set	mip	ordtype	*i*	
set	mip strategy	backtrack		page 158, page 176
set	mip strategy	bbinterval		page 158
set	mip strategy	branch	*i*	page 158
set	mip strategy	crossover	*i*	page 188
set	mip strategy	file	*i*	
set	mip strategy	heuristicfreq		page 163, page 176
set	mip strategy	mipstart	1	page 165
set	mip strategy	nodeselect	*i*	page 158, page 177
set	mip strategy	order	*i*	page 163
set	mip strategy	presolvenode		
set	mip strategy	probe		page 175
set	mip strategy	startalgorithm		page 187
set	mip strategy	subalgorithm	*i*	page 186, page 187, page 188
set	mip strategy	variableselect	*i*	page 158, page 176, page 178
set	mip tolerances	absmipgap	3.0	page 179
set	mip tolerances	integrality	*i*	

Command		Options		Example
set	mip tolerances	lowercutoff	*n*	page 157, page 180
set	mip tolerances	mipgap	0.01	page 179
set	mip tolerances	objdifference	*n*	page 157, page 179
set	mip tolerances	relobjdifference	*n*	page 157, page 180
set	mip tolerances	uppercutoff	*n*	page 157, page 180
set	network	display	*i*	page 223
set	network	iterations	*i*	page 224
set	network	netfind	*i*	page 238
set	network	pricing	*i*	page 224
set	network tolerances	feasibility	*i*	page 223
set	network tolerances	optimality	*i*	
set	output	logonly	*y*	page 114, page 116
set	output	channel	v1 v2	
set	preprocessing	aggregator	2	page 98, page 108, page 164
set	preprocessing	boundstrength		page 164
set	preprocessing	coeffreduce		page 164, page 189
set	preprocessing	dependency	*1*	page 98, page 141
set	preprocessing	dual	*1*	page 100, page 143
set	preprocessing	fill	*i*	page 98
set	preprocessing	linear	*i*	

Command		Options		Example
`set`	`preprocessing`	`numpass`	*i*	
`set`	`preprocessing`	`presolve`	*i*	page 98, page 100, page 108, page 164
`set`	`preprocessing`	`reduce`	*i*	page 98, page 112
`set`	`preprocessing`	`relax`		page 164
`set`	`read`	`constraints`	*i j*	
`set`	`read`	`datacheck`	*i*	
`set`	`read`	`nonzeroes`	*i j*	
`set`	`read`	`qpnonzeroes`	*i j*	
`set`	`read`	`reverse`	*i*	
`set`	`read`	`scale`	*i*	page 105
`set`	`read`	`variables`	*i j*	
`set`	`simplex`	`basisinterval`	*i*	
`set`	`simplex`	`crash`	*i*	page 106
`set`	`simplex`	`dgradient`	*i*	page 104, page 105
`set`	`simplex`	`display`	*i*	
`set`	`simplex`	`iisfind`	*i*	
`set`	`simplex limits`	`iterations`	*i*	
`set`	`simplex limits`	`lowerobj`	*i*	
`set`	`simplex limits`	`perturbation`	*i*	page 110
`set`	`simplex limits`	`singularity`	*i*	page 109
`set`	`simplex limits`	`upperobj`		
`set`	`simplex`	`perturbation`	`1 i`	page 110
`set`	`simplex`	`pgradient`	*i*	page 104

Command		Options		Example
`set`	`simplex`	`pricing`		
`set`	`simplex`	`refactor`	*`i`*	page 105, page 107
`set`	`simplex tolerances`	`feasibility`	*`n`*	page 114
`set`	`simplex tolerances`	`markowitz`	*`n`*	page 111
`set`	`simplex tolerances`	`optimality`	*`n`*	page 114
`set`	`simplex`	`xxxstart`		
`set`	`timelimit`			page 166
`set`	`workdir`	`directory`		
`set`	`workmem`	`filesize`		page 166, page 185
`tranopt`				page 96
`write`	*`filename`*	*`type`*		page 98, page 109, page 118, page 186, page 132, page 241
`xecute`	`command`			page 118

Managing Parameters in the Interactive Optimizer

To see the current value of a parameter that interests you in the Interactive Optimizer, use the command `display settings`. The command `display settings changed` lists only those parameters whose values are not the default value. The command `display settings all` lists all parameters and their values.

To change the value of a parameter in the Interactive Optimizer, use the command `set` followed by options to indicate the parameter and the value you want it to assume.

The *ILOG CPLEX Reference Manual* documents the name of each parameter and its options in the Interactive Optimizer.

Saving a Parameter Specification File

You can tell the ILOG CPLEX Interactive Optimizer to read customized parameter settings from a *parameter specification file*. By default, ILOG CPLEX expects a parameter specification file to be named `cplex.par`, and it looks for that file in the directory where it is executing. However, you can rename the file, and you can tell ILOG CPLEX to look for it in another directory by setting the system environment variable `CPLEXPARFILE` to the full path name (including a new name, a new location) of your parameter specification file. You set that environment variable in the customary way for your platform. For example, on a Unix platform, you might use a shell command to set the environment variable, or on a personal computer running NT, you might click on the System icon in the control panel, then select the environment tab from the available system properties tabs.

During initialization in the Interactive Optimizer, ILOG CPLEX locates any available parameter specification file (by checking the current execution directory for `cplex.par` and by checking the environment variable `CPLEXPARFILE`) and reads that file. As it opens the file, ILOG CPLEX displays the message “Initial parameter values are being read from `cplex.par`” (or from the parameter specification file you specified). As ILOG CPLEX displays that message on the screen, it also writes the message to the log file. If ILOG CPLEX cannot open the file, it displays no message, records no note in the log file, and uses default parameter settings.

You can use a parameter specification file to change any parameter or parameters accessible by the `set` command in the Interactive Optimizer. The parameter types, names, and options are those used by the `set` command in the Interactive Optimizer.

To create a parameter specification file, you can use either of these alternatives:

◆ Use an ordinary text editor to create a file where each line observes the following syntax:

  parameter-name option value

◆ Use the command `display settings` in the Interactive Optimizer to generate a list of current parameter settings. Those settings will be recorded in the log file. You can then edit the log file to create your parameter specification file.

  `display settings changed` lists parameters different from the default.

  `display settings all` lists all parameters.

Each entry on a line must be separated by at least one space or tab. Blank lines in a parameter specification file are acceptable; there are no provisions for comments in the file. You may abbreviate parameter names to unique character sequences, as you do in the `set` command.

As ILOG CPLEX reads a parameter specification file, if the parameter name and value are valid, ILOG CPLEX sets the parameter and writes a message about it to the screen and to the log file. If ILOG CPLEX encounters a repeated parameter, it uses the last value specified. ILOG CPLEX terminates under the following conditions:

- if it encounters a parameter that is unknown;
- if it encounters a parameter that is not unique;
- if the parameter is correctly specified but the value is missing, invalid, or out of range.

Here is an example of a parameter specification file that resets the limits on the size of problem reads and opens a log file named `problem.log`.

```
read constraints    50
read variables     100
read nonzeros      500
logfile            problem.log
```

Index