IBM ILOG CPLEX Optimization Studio Getting Started with CPLEX

Version 12 Release 6

Copyright notice

Describes general use restrictions and trademarks related to this document and the software described in this document.

© Copyright IBM Corp. 1987, 2013

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Trademarks

IBM, the IBM logo, ibm.com, WebSphere, and ILOG are trademarks or registered trademarks of International Business Machines Corp., in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.

Further acknowledgments

IBM ILOG CPLEX states these additional registered trademarks, copyrights, and acknowledgments.

Additional registered trademarks, copyrights, licenses

Python is a registered trademark of the Python Software Foundation.

MATLAB is a registered trademark of The MathWorks, Inc.

OpenMPI is distributed by The Open MPI Project under the New BSD license and copyright 2004 - 2012.

MPICH2 is copyright 2002 by the University of Chicago and Argonne National Laboratory.

Acknowledgment of use: dtoa routine of the gdtoa package

IBM ILOG CPLEX acknowledges use of the `dtoa` routine of the `gdtoa` package, available at

`http://www.netlib.org/fp/`.

The author of this software is David M. Gay.

All Rights Reserved.

Copyright (C) 1998, 1999 by Lucent Technologies

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the name of Lucent or any of its entities not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

(end of acknowledgment of use of `dtoa` routine of the `gdtoa` package)

© Copyright IBM Corporation 1987, 2013.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Introducing CPLEX . . . v

What is CPLEX? . . . v
- Types of problems solved . . . v
- CPLEX components . . . vi
- Optimizer options . . . vii

Parallel optimizers . . . vii
Data entry options . . . viii
What CPLEX is not . . . viii
What you need to know . . . viii
What's in this manual . . . ix
Notation in this manual . . . ix
Related documentation . . . x

Chapter 1. Setting up CPLEX . . . 1

Installing CPLEX . . . 1
Setting up CPLEX on Windows . . . 1
Setting up CPLEX on GNU/Linux . . . 2
Setting up Eclipse for the Java API of CPLEX . . . 4
Setting up the Python API of CPLEX . . . 5
Directory structure of CPLEX . . . 6
Using the Component Libraries . . . 7

Chapter 2. Solving an LP with CPLEX 11

Overview . . . 11
Problem statement . . . 11
Using the Interactive Optimizer . . . 11
Using Concert Technology in C++ . . . 12
Using Concert Technology in Java . . . 13
Using Concert Technology in .NET . . . 13
Using the Callable Library . . . 14
Using the Python API . . . 16

Chapter 3. Interactive Optimizer tutorial 19

Starting CPLEX . . . 19
Using help . . . 19
Entering a problem . . . 21
- Overview . . . 21
- Entering the example . . . 21
- Using the LP format . . . 22
- Entering data . . . 24

Displaying a problem . . . 24
- Verifying a problem with the display command 24
- Displaying problem statistics . . . 25
- Specifying item ranges . . . 26
- Displaying variable or constraint names . . . 27
- Ordering variables . . . 28
- Displaying constraints . . . 28
- Displaying the objective function . . . 28
- Displaying bounds . . . 28
- Displaying a histogram of nonzero counts . . . 29

Solving a problem . . . 29
- Where you are . . . 30
- Solving the example . . . 30
- Solution options . . . 31
- Displaying post-solution information . . . 32

Performing sensitivity analysis . . . 33
Writing problem and solution files . . . 34
- Overview . . . 34
- Selecting a write file format . . . 34
- Writing LP files . . . 35
- Writing basis files . . . 35
- Using path names . . . 36

Reading problem files . . . 36
- Overview . . . 36
- Selecting a read file format . . . 36
- Reading LP files . . . 37
- Using file extensions . . . 38
- Reading MPS files . . . 38
- Reading basis files . . . 38

Setting CPLEX parameters . . . 39
Adding constraints and bounds . . . 40
Changing a problem . . . 41
- Overview . . . 41
- What can be changed? . . . 42
- Changing constraint or variable names . . . 42
- Changing sense . . . 42
- Changing bounds . . . 43
- Removing bounds . . . 43
- Changing coefficients of variables . . . 43
- Objective and RHS coefficients . . . 44
- Deleting entire constraints or variables . . . 44
- Changing small values to zero . . . 45

Executing operating system commands . . . 45
Quitting CPLEX . . . 46
Advanced features of the Interactive Optimizer . . . 46

Chapter 4. Concert Technology tutorial for C++ users . . . 49

The design of CPLEX in Concert Technology C++ applications . . . 49
Compiling CPLEX in Concert Technology C++ applications . . . 50
- Testing your installation on UNIX . . . 50
- Testing your installation on Windows . . . 50
- In case of problems . . . 50

The anatomy of a Concert Technology C++ application . . . 51
- Constructing the environment: IloEnv . . . 51
- Creating a model: IloModel . . . 52
- Solving the model: IloCplex . . . 54
- Querying results . . . 54
- Handling errors . . . 55

Building and solving a small LP model in C++ . . . 55
- Overview . . . 56
- Modeling by rows . . . 57
- Modeling by columns . . . 57
- Modeling by nonzero elements . . . 58

Writing and reading models and files . . . 58
Selecting an optimizer . . . 59
Reading a problem from a file: example ilolpex2.cpp 60

© Copyright IBM Corp. 1987, 2013

Overview . . . 60
Reading the model from a file . . . 60
Selecting the optimizer . . . 60
Accessing basis information . . . 61
Querying quality measures . . . 61
Modifying and re-optimizing . . . 61
Modifying an optimization problem: example ilolpex3.cpp . . . 62
Overview . . . 62
Setting CPLEX parameters . . . 63
Modifying an optimization problem . . . 63
Starting from a previous basis . . . 63
Complete program . . . 63

Chapter 5. Concert Technology tutorial for Java users . . . 65
Overview . . . 65
Compiling CPLEX in Concert Technology Java applications . . . 65
Paths and JARs . . . 65
Adapting build procedures to your platform . . . 65
In case problems arise . . . 66
The design of CPLEX in Concert Technology Java applications . . . 67
The anatomy of a Concert Technology Java application . . . 67
Structure of an application . . . 67
Create the model . . . 68
Solve the model . . . 69
Query the results . . . 70
Building and solving a small LP model in Java . . . 70
Example: LPex1.java . . . 70
Modeling by rows . . . 72
Modeling by columns . . . 72
Modeling by nonzeros . . . 73

Chapter 6. Concert Technology tutorial for .NET users . . . 75
Presenting the tutorial . . . 75
What you need to know: prerequisites . . . 75
What you will be doing . . . 76
Describe . . . 77
Model . . . 78
Solve . . . 81
Complete program . . . 82

Chapter 7. Callable Library tutorial . . . 83
The design of the CPLEX Callable Library . . . 83
Compiling and linking Callable Library applications 83
Overview . . . 83
Building Callable Library applications on UNIX platforms . . . 84
Building Callable Library applications on Win32 platforms . . . 84
How CPLEX works . . . 85
Overview . . . 85
Opening the CPLEX environment . . . 85
Instantiating the problem object . . . 86
Populating the problem object . . . 86
Changing the problem object . . . 86
Creating a successful Callable Library application 87
Overview . . . 87
Prototype the model . . . 87
Identify the routines to call . . . 87
Test procedures in the application . . . 87
Assemble the data . . . 88
Choose an optimizer . . . 88
Observe good programming practices . . . 89
Debug your program . . . 89
Test your application . . . 89
Use the examples . . . 89
Building and solving a small LP model in C . . . 90
Reading a problem from a file: example lpex2.c . . . 91
Adding rows to a problem: example lpex3.c . . . 92
Performing sensitivity analysis . . . 94

Chapter 8. Python tutorial . . . 97
Design of CPLEX in a Python application . . . 97
Starting the CPLEX Python API . . . 97
Accessing the module `cplex` . . . 97
Building and solving a small LP with Python . . . 98
Reading and writing CPLEX models to files with Python . . . 98
Selecting an optimizer in Python . . . 99
Example: reading a problem from a file lpex2.py 100
Modifying and re-optimizing in the CPLEX Python API. . . . 100
Example: modifying a model lpex3.py . . . 101
Using CPLEX parameters in the CPLEX Python API. . . . 102

Index . . . 105

Introducing CPLEX

This preface introduces CPLEX.

What is CPLEX?

CPLEX consists of software components and options.

Types of problems solved

Defines the kind of problems that CPLEX solves.

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

Maximize (or Minimize)	$c_1 x_1 + c_2 x_2 +...+ c_n x_n$
subject to	$a_{11} x_1 + a_{12} x_2 +...+ a_{1n} x_n \sim b_1$
	$a_{21} x_1 + a_{22} x_2 +...+ a_{2n} x_n \sim b_2$
	...
	$a_{m1} x_1 + a_{m2} x_2 +...+ a_{mn} x_n \sim b_m$
with these bounds	$l_1 \leq x_1 \leq u_1$
	...
	$l_n \leq x_n \leq u_n$

where ~ can be ≤, ≥, or =, and the upper bounds u_i and lower bounds l_i may be positive infinity, negative infinity, or any real number.

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

Objective function coefficients	$c_1, c_2, \ldots, c_n$
Constraint coefficients	$a_{11}, a_{21}, \ldots, a_{n1}$
	...
	$a_{m1}, a_{m2}, \ldots, a_{mn}$
Righthand sides	$b_1, b_2, \ldots, b_m$
Upper and lower bounds	$u_1, u_2, \ldots, u_n$ *and* $l_1, l_2, \ldots, l_n$

The optimal solution that CPLEX computes and returns is:

Variables	$x_1, x_2, \ldots, x_n$

CPLEX also can solve several extensions to LP:

- Network Flow problems, a special case of LP that CPLEX can solve much faster by exploiting the problem structure.

© Copyright IBM Corp. 1987, 2013

- Quadratic Programming (QP) problems, where the LP objective function is expanded to include quadratic terms.
- Quadratically Constrained Programming (QCP) problems that include quadratic terms among the constraints. In fact, CPLEX can solve Second Order Cone Programming (SOCP) problems.
- Mixed Integer Programming (MIP) problems, where any or all of the LP, QP, or QCP variables are further restricted to take integer values in the optimal solution and where MIP itself is extended to include constructs like Special Ordered Sets (SOS) and semi-continuous variables.

CPLEX components

Describes the components of CPLEX: Interactive Optimizer, Concert Technology, Callable Library.

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

- The **CPLEX Interactive Optimizer** is an executable program that can read a problem interactively or from files in certain standard formats, solve the problem, and deliver the solution interactively or into text files. The program consists of the file `cplex.exe` on Windows platforms or `cplex` on UNIX platforms.
- **Concert Technology** is a set of C++, Java, and .NET class libraries offering an API that includes modeling facilities to allow the programmer to embed CPLEX optimizers in C++, Java, or .NET applications. Table 1. lists the files that contain the libraries.

Table 1. Concert Technology libraries

	Microsoft Windows	UNIX
C++	`ilocplex.lib concert.lib`	`libilocplex.a libconcert.a`
Java	`cplex.jar`	`cplex.jar`
.NET	`ILOG.CPLEX.dll` `ILOG.Concert.dll`	

  The Concert Technology libraries make 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, FORTRAN, or any other language that can call C functions. The library is provided in files `cplexXXX.lib` and `cplexXXX.dll` on Windows platforms, and in `libcplex.a`, `libcplex.so`, and `libcplex.sl` on UNIX platforms.
- The **Python API** for CPLEX a full-featured Python application programming interface supporting all aspects of CPLEX optimization.
- The **CPLEX connector for The MathWorks MATLAB** enables a user to define optimization problems and solve them within MATLAB using either the MATLAB Toolbox or a CPLEX class in the MATLAB language.

In this manual, the phrase *CPLEX Component Libraries* is used to refer equally to any of these libraries. While all of the libraries are callable, the term *CPLEX Callable Library* as used here refers specifically to the C library.

Compatible platforms

CPLEX is available on Windows, UNIX, and other platforms. The programming interface works the same way and provides the same facilities on all platforms.

Installation

If you have not yet installed CPLEX on your platform, consult Chapter 1, "Setting up CPLEX," on page 1. It contains instructions for installing CPLEX.

Optimizer options

Introduces the options available in CPLEX.

This manual explains how to use the LP algorithms that are part of CPLEX. The QP, QCP, and MIP problem types are based on the LP concepts discussed here, and the extensions to build and solve such problems are explained in the *CPLEX User's Manual*.

Default settings will result in a call to an optimizer that is appropriate to the class of problem you are solving. However, you may wish to choose a different optimizer for special purposes. An LP or QP problem can be solved using any of the following CPLEX optimizers: dual simplex, primal simplex, barrier, and perhaps also the network optimizer (if the problem contains an extractable network substructure). Pure network models are all solved by the network optimizer. QCP models, including the special case of SOCP models, are all solved by the barrier optimizer. MIP models are all solved by the mixed integer optimizer, which in turn may invoke any of the LP or QP optimizers in the course of its computation. The table titled Table 2 summarizes these possible choices.

Table 2. Optimizers

	LP	Network	QP	QCP	MIP
Dual Optimizer	yes		yes		
Primal Optimizer	yes		yes		
Barrier Optimizer	yes		yes	yes	
Mixed Integer Optimizer					yes
Network Optimizer	Note 1	yes	Note 1		
Note 1: The problem must contain an extractable network substructure.					

The choice of optimizer or other parameter settings may have a very large effect on the solution speed of your particular class of problem. The *CPLEX User's Manual* describes the optimizers, provides suggestions for maximizing performance, and notes the features and algorithmic parameters unique to each optimizer.

Parallel optimizers

Parallel optimizers are available in CPLEX.

Parallel barrier, parallel MIP, and concurrent optimizers are implemented to run on hardware platforms with parallel processors. These parallel optimizers can be called from the Interactive Optimizer and the Component Libraries.

When small models, such as those in this document, are being solved, the effect of parallelism will generally be negligible. On larger models, the effect is ordinarily beneficial to solution speed.

See the topic Parallel optimizers in the *CPLEX User's Manual* for information about using CPLEX on a parallel computer.

Data entry options

CPLEX supports a variety of data entry options.

CPLEX provides several options for entering your problem data. When using the Interactive Optimizer, most users will enter problem data from formatted files. CPLEX supports the industry-standard MPS (Mathematical Programming System) file format as well as CPLEX LP format, a row-oriented format many users may find more natural. Interactive entry (using CPLEX LP format) is also a possibility for small problems.

Data entry options are described briefly in this manual. File formats are documented in the reference manual, File formats supported by CPLEX.

Concert Technology and Callable Library users may read problem data from the same kinds of files as in the Interactive Optimizer, or they may want to pass data directly into CPLEX to gain efficiency. These options are discussed in a series of examples that begin with "Building and solving a small LP model in C++" on page 55, "Building and solving a small LP model in Java" on page 70, and "Building and solving a small LP model in C" on page 90 for the CPLEX Callable Library users.

Users can also read models from Python. For more about that approach, see the topic "Reading and writing CPLEX models to files with Python" on page 98 in the tutorial for Python users in this manual.

Users can also read models from The MathWorks MATLAB. For more about that approach, see the user's manual accompanying the CPLEX connector for MATLAB.

What CPLEX is not

CPLEX contrasts with other tools, such as modeling languages or integrated development environments.

CPLEX Optimizer is not a modeling language, nor is it an integrated development environment (IDE). You can completely model and solve your optimization problems with CPLEX; however, the optimizer that it provides does not offer the interactive facilities of a modeling system in an integrated development environment. For such features as interactive modeling, consider the integrated development environment of IBM ILOG CPLEX Optimization Studio.

What you need to know

Prerequisites for effective use of CPLEX include familiarity with your operating system, knowledge of file management, and facility in a programming language.

In order to use CPLEX effectively, you need to be familiar with your operating system.

This manual 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 available for users of C++, Java, and the .NET framework. 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.

What's in this manual

Getting Started with CPLEX offers tutorials for the components of CPLEX, including the Interactive Optimizer and the application programming interfaces.

Chapter 1, "Setting up CPLEX," on page 1 tells how to install CPLEX.

Chapter 2, "Solving an LP with CPLEX," on page 11 shows you at a glance how to use the Interactive Optimizer and each of the application programming interfaces (APIs): C++, Java, .NET, and C. This overview is followed by more detailed tutorials about each interface.

Chapter 3, "Interactive Optimizer tutorial," on page 19 explains, step by step, how to use the Interactive Optimizer: how to start it, how to enter problems and data, how to read and save files, how to modify objective functions and constraints, and how to display solutions and analytical information.

Chapter 4, "Concert Technology tutorial for C++ users," on page 49 describes the same activities using the classes in the C++ implementation of the CPLEX Concert Technology Library.

Chapter 5, "Concert Technology tutorial for Java users," on page 65 describes the same activities using the classes in the Java implementation of the CPLEX Concert Technology Library.

Chapter 6, "Concert Technology tutorial for .NET users," on page 75 describes the same activities using .NET facilities.

Chapter 7, "Callable Library tutorial," on page 83, describes the same activities using the routines in the CPLEX Callable Library.

Chapter 8, "Python tutorial," on page 97 covers certain installation considerations plus an introduction to using the CPLEX Python API to model and solve optimization problems.

All tutorials use examples that are delivered with the standard distribution.

Notation in this manual

Notation in this manual conforms to familiar conventions.

This manual observes the following conventions in notation and names.

- Important ideas are *emphasized* the first time they appear.

- Text that is entered at the keyboard or displayed on the screen as well as commands and their options available through the Interactive Optimizer appear in `this typeface`, for example, `set preprocessing aggregator n`.
- Entries that you must fill in appear in *this typeface*; for example, `write` *filename*.
- The names of C routines and parameters in the CPLEX Callable Library begin with `CPX` and appear in `this typeface`, for example, `CPXcopyobjnames`.
- The names of C++ classes in the CPLEX Concert Technology Library begin with `Ilo` and appear in `this typeface`, for example, `IloCplex`.
- The names of Java classes begin with `Ilo` and appear in `this typeface`, for example, `IloCplex`.
- The name of a class or method in .NET is written as concatenated words with the first letter of each word in upper case, for example, `IntVar` or `IntVar.VisitChildren`. Generally, accessors begin with the key word `Get`. Accessors for Boolean members begin with `Is`. Modifiers begin with `Set`.
- Combinations of keys from the keyboard are hyphenated. For example, control-c indicates that you should press the control key and the c key simultaneously. The symbol `<return>` indicates end of line or end of data entry. On some keyboards, the key is labeled enter or Enter.

Related documentation

Additional documentation is available for CPLEX.

In addition to this introductory manual, the standard distribution of CPLEX comes with the *CPLEX User's Manual* and the *CPLEX Reference Manuals*. All documentation is available online. It is delivered with the standard distribution of the product and accessible through conventional HTML browsers for customers on most platforms.

- The *CPLEX User's Manual* explains the relationship between the Interactive Optimizer and the Component Libraries. It enlarges on aspects of linear programming with CPLEX and shows you how to handle quadratic programming (QP) problems, quadratically constrained programming (QCP) problems, second order cone programming (SOCP) problems, and mixed integer programming (MIP) problems. It tells you how to control CPLEX parameters, debug your applications, and efficiently manage input and output. It also explains how to use parallel CPLEX optimizers.
- The *CPLEX Callable Library Reference Manual* documents the Callable Library routines and their arguments. This manual also includes additional documentation about error codes, solution quality, and solution status.
- The *CPLEX C++ API Reference Manual* documents the C++ API of the Concert Technology classes, methods, and functions.
- The *CPLEX Java API Reference Manual* supplies detailed definitions of the Concert Technology interfaces and CPLEX Java classes.
- The *CPLEX .NET Reference Manual* documents the .NET API for CPLEX.
- The *CPLEX Python API Reference Manual* documents the Python API for CPLEX.
- The reference manual *CPLEX Parameters* contains documentation of parameters that can be modified by parameter routines. It is the definitive reference manual for the purpose and allowable settings of CPLEX parameters.
- The reference manual *CPLEX File Formats* contains a list of file formats that CPLEX supports as well as details about using them in your applications.

- The reference manual *CPLEX Interactive Optimizer* contains the commands of the Interactive Optimizer, along with the command options and links to examples of their use in the *CPLEX User's Manual.*
- A suite of documentation, including a user's manual and language reference manual, is available for the *CPLEX connector for MATLAB*. This documentation is available either interactively within a MATLAB session or online.

As you work with CPLEX on a long-term basis, you should read the complete *User's Manual* to learn how to design models and implement solutions to your own problems. Consult the reference manuals for authoritative documentation of the Component Libraries, their application programming interfaces (APIs), and the Interactive Optimizer.

Chapter 1. Setting up CPLEX

To set up CPLEX for your particular platform or integrated development environment, and to check your set up, follow these steps.

Installing CPLEX

CPLEX is installed as a feature of IBM ILOG CPLEX Optimization Studio.

After you successfully install IBM ILOG CPLEX Optimization Studio, all of the facilities of CPLEX, a feature of IBM ILOG CPLEX Optimization Studio, become functional and are available to you.

This topic directs you to more information about settings specific to your platform or integrated development environment.

- "Setting up CPLEX on Windows"
- "Setting up Eclipse for the Java API of CPLEX" on page 4
- "Setting up the Python API of CPLEX" on page 5

Other topics provide Tutorials in the use of each of the components that CPLEX provides: the Interactive Optimizer tutorial, the Concert Technology tutorials for C++, Java, and .NET users, the Callable Library tutorial for C and other languages callable from C, as well as the tutorials for users of Python or the Microsoft Solver Foundation (MSF). More extensive documentation for the CPLEX connector to Microsoft Excel and for the CPLEX connector to MATLAB is available as online help inside sessions of those products.

Important:

Remember that most distributions of the product operate correctly only on the specific platform and operating system for which they are designed. If you upgrade your operating system, you may need to obtain a new distribution of the product. In that case, contact your IBM representative for advice.

Setting up CPLEX on Windows

You can customize your installation of CPLEX for use on Microsoft Windows.

Normally, the installer you receive from IBM installs CPLEX correctly for you; this topic highlights customization for users of Microsoft Windows.

Tip:

The CPLEX library on Windows is a DLL file. The name of the CPLEX DLL looks like this: `cplexXXX.dll` where `XXX` represents the current version number.

Make sure that your application has read and execute permission for both the CPLEX DLL and the directory where it is located. In other words, set the correct authorizations on both the CPLEX DLL and the folder where it resides for your application to read and execute.

© Copyright IBM Corp. 1987, 2013

After your installation of IBM ILOG CPLEX Optimization Studio, if your applications are not able to find the DLL for CPLEX at runtime, then you may need to identify its location for them. On Windows, there are alternative ways to do so, either through the operating system or in an integrated development environment (IDE).

- At the operating system, you can add the location of `cplexXXX.dll` to the environment variable `PATH`.

  After you extend the environment variable at the level of the operating system like this, all applications that are aware of the PATH environment variable will know where to find your CPLEX DLL. The topic "Adding your CPLEX DLL to the environment variable on Windows" explains this alternative in detail.
- In Visual Studio, you can link the location of `cplexXXX.dll` to the properties of the project.

  After you add your CPLEX DLL to the Linker Path of your Visual Studio project like this, your Project will know where to find your CPLEX DLL. The topic "Linking your CPLEX DLL to the properties of your project in Visual Studio" explains this alternative in detail.

Adding your CPLEX DLL to the environment variable on Windows

1. From the Start menu, select Control Panel.
2. In the Control Panel, select System.
3. In the System dialog, select the Advanced tab.
4. On the Advanced tab, click the Environment Variables button.
5. Add or extend the `PATH` environment variable. If the `PATH` environment variable already exists, extend it, like this:

   ```
   Name: PATH
   Value: %PATH%;C:\yourCPLEXhome\CPLEXXXX\bin\x86_win32
   ```

   where `XXX` represents the current version number and `yourCPLEXhome` represents the folder where you installed CPLEX.
6. Restart Visual Studio and other applications for this change in the operating system to take effect.

Linking your CPLEX DLL to the properties of your project in Visual Studio

1. In Visual Studio, right-click your project.
2. Select Properties of the project.
3. Among the Properties of the project, go to the Linker section.
4. In the Linker section, add the name of the folder containing the CPLEX DLL file to the Linker Path.
5. Save the properties of your project.

Setting up CPLEX on GNU/Linux

You can customize your installation of CPLEX for C or C++ applications.

Normally, the installer you receive from IBM installs CPLEX correctly for you. This topic highlights customization by users of GNU/Linux, UNIX, and similar operating systems for C and C++ applications. For similar hints about setting up for Java applications, see the topic "Setting up Eclipse for the Java API of CPLEX" on page 4. Likewise, for hints about setting up for Python applications, see the

topic "Setting up the Python API of CPLEX" on page 5. For hints about setting up for .NET applications in Microsoft Visual Studio, see the topic "Linking your CPLEX DLL to the properties of your project in Visual Studio" on page 2.

CPLEX is delivered with an assortment of sample make files that accompany the examples delivered with the product. As you design your own make files for your CPLEX applications, it is a good idea to start from one of those sample make files and adapt it as needed.

If you are writing a make file "from scratch" consider these questions:

- Where did you install CPLEX? In this topic, for purposes of illustration, assume the product was installed in `/path/to/cplex/cplexXX/` where `XX` designates the version number.
- What is your architecture? That is, what is your combination of operating system and compiler? In this topic, for purposes of illustration, assume the architecture is `archi/tecture`.

With answers to those questions, now consider options for your **compiler** and **linker**. These options vary slightly according to whether you target C or C++ applications because C++ applications need the Concert include files and library as well as the CPLEX include files and library.

Compiler options

For **C** applications:

- Tell your compiler where to find the `include` directory for CPLEX Callable Library (C API) by means of the option `-I` and the path to the C header files, like this, substituting the correct path of your installation and the correct version number:
  ```
  -I/path/to/cplex/cplexXX/include
  ```
- Give your compiler a preprocessor directive by means of the option `-D`, like this:
  ```
  -DIL_STD
  ```

For **C++** applications:

- Tell your compiler where to find the include directory for both CPLEX C++ API and Concert C++ API, substituting the correct path of your installation and the correct version number, like this:
  ```
  -I/path/to/cplex/cplexXX/include
  -I/path/to/cplex/concertYY/include
  ```
- Give your compiler a preprocessor directive by means of the option `-D`, like this:
  ```
  -DIL_STD
  ```

Linker options

For **C** applications:

- Tell your linker where to search for the library, like this, substituting the correct path of your installation and the correct version number:
  ```
  -L/path/to/cplex/cplexXX/lib/archi/tecture
  ```
- Specify the library to your linker, like this:
  ```
  -lcplex -lm -lpthread
  ```

For **C++** applications:

- Tell your linker where to search for both the CPLEX and Concert libraries, like this, substituting the correct path of your installation and the correct version numbers:

  ```
  -L/path/to/cplex/cplexXX/lib/archi/tecture
  -L/path/to/cplex/concertYY/lib/archi/tecture
  ```

- Specify the libraries to your linker, like this:

  ```
  -lilocplex -lconcert
  ```

  Here is the correct order of the builds of the libraries:

  ```
  -lilocplex -lconcert -lcplex -lm -lpthread
  ```

Setting up Eclipse for the Java API of CPLEX

Regardless of platform, you can make Eclipse aware of the Java API of CPLEX.

To make your Eclipse integrated development environment aware of the Java API of CPLEX, follow these steps.

1. Either import an existing project, or create a new Java project.

   To import an existing Java project, first click **File** > **Import ...**. When **Select** opens, expand the **General** folder. Within the **General** folder, select **Existing Projects into Workspace**.

   To create a new Java project, click **File** > **New ...** > **Java Project**.

2. To include the CPLEX JAR in your project, locate the libraries among the properties of the project, like this:

   Project > **Properties** > **Java Build Path** > **Libraries**

3. Click the button **Add External JARs**.
4. Browse to the location of your CPLEX installation and select the file named `cplex.jar`.

Tip:

Be sure to point to the CPLEX JAR, `cplex.jar`, not simply the folder or directory containing that file. Conventionally, the CPLEX JAR is in the `lib` directory of your CPLEX home, part of the installation of IBM ILOG CPLEX Optimization Studio.

After that procedure, you can create and compile your source code making use of the Java API of CPLEX.

Before you can run your compiled code, you must create an appropriate **run configuration** in Eclipse or modify an existing **run configuration** appropriately. To do so, follow these steps.

1. Create a new run configuration or locate an existing run configuration to modify, like this:

   Run > Run Configurations ... > Java Application

2. When the dialog appears, go to the **Main** tab, and select your `main` class; that is, select the class containing the `main()` function that you want to run.
3. Go to the **Arguments** tab, select **VM arguments**, and add the path to the CPLEX library. The CPLEX library is conventionally a DLL file on Windows or a .so file on UNIX or GNU/Linux, with an indication of the version number in its name, like `cplexXX.dll` or `cplexXX.so` where XX represents a version number of the product.

   Tip:

Add the path to the directory or folder containing the CPLEX library (**not** the name of the file itself).

For example, on UNIX or GNU/Linux , if the file `libcplexXX.so` is located in the directory `COSinstallation/cplex/bin/myPlatform/myLibFormat` then add a path similar to this example:

```
-Djava.library.path= COSinstallation/cplex/myPlatform/myLibFormat
```

Similarly, on Windows, if the file `cplexXX.dll` is located in the folder `COSinstallation\cplex\myPlatform\myLibFormat` then add a path similar to this example:

```
-Djava.library.path=COSinstallation/cplex/myPlatform/myLibFormat
```

After these steps, you can run your Java application using CPLEX as you run other applications from Eclipse.

Setting up the Python API of CPLEX

The Python API of CPLEX is part of IBM ILOG CPLEX Optimization Studio.

The modules associated with the CPLEX Python API reside in the directory `yourCPLEXhome/python/PLATFORM`, (or in the folder `yourCPLEXhome\python\PLATFORM`) where `yourCPLEXhome` specifies the location where CPLEX is installed as part of IBM ILOG CPLEX Optimization Studio, and `PLATFORM` stands for your combination of operating system and compiler.

There are two alternative ways to set up the Python API of CPLEX.

- The most common way is to use the script `setup.py` located in the directory `yourCPLEXhome/python/PLATFORM` (or in the folder `yourCPLEXhome\python\PLATFORM`).
- Alternatively, you can set the environment variable `PYTHONPATH` to `yourCPLEXhome/python/PLATFORM` and start running Python scripts with CPLEX.

Both of these methods are detailed further in the following paragraphs.

Using the script setup.py

To install the CPLEX-Python modules on your system, use the script `setuy.py` located in `yourCplexhome/python/PLATFORM`. If you want to install the CPLEX-Python modules in a nondefault location, use the option `--home` to identify the installation directory. For example, to install the CPLEX-Python modules in the default location, use the following command from the command line:

```
python setup.py install
```

To install in the directory `yourPythonPackageshome/cplex`, use the following command from the command line:

```
python setup.py install --home yourPythonPackageshome/cplex
```

Both of those commands (default and home-specified) invoke the Python package `distutils`. For other options available with that package, consult the documentation of Python `distutils`.

Setting the environment variable PYTHONPATH

If you maintain multiple versions of CPLEX, or if you run multiple versions of CPLEX side-by-side, then use this way of declaring the location of CPLEX and its Python API to your Python installation by means of the environment variable `PYTHONPATH`.

To start using the CPLEX Python API, set the Python path environment variable `PYTHONPATH` to the value of `yourCplexhome/python/PLATFORM`. Setting this environment variable enables Python to find the CPLEX modules that it needs to run Python commands and scripts that use the CPLEX Python API.

Next steps

After setting up your Python environment by means of one of those alternative methods, you can proceed to the topic "Starting the CPLEX Python API" on page 97.

Directory structure of CPLEX

Your CPLEX home directory is part of your IBM ILOG CPLEX Optimization Studio installation.

After you install IBM ILOG CPLEX Optimization Studio, you find CPLEX in a structure (such as folders or directories) like the one in the illustrations Figure 1 on page 7 and Figure 2 on page 7. This structure is relative to the location where you installed IBM ILOG CPLEX Optimization Studio as your COS home, containing your CPLEX home.

In these illustrations, `platform` specifies the combination of operating system and chip architecture (such as 32- or 64-bit) Likewise, `libformat` specifies library format and compiler options, such as static, position-independent, and so forth.

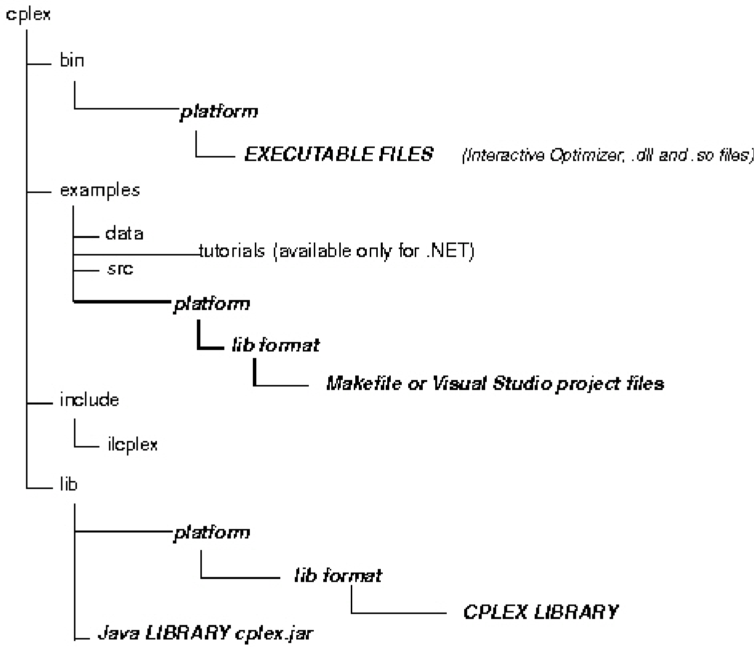

Figure 1. Structure of the CPLEX directory

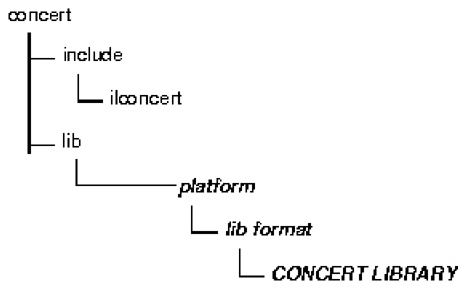

Figure 2. Structure of the Concert Technology Directory

Using the Component Libraries

Examples provided with CPLEX help you check your installation.

After you have completed the installation, you can verify that everything is working by running one or more of the examples that are provided with the standard distribution.

Verifying set-up on UNIX

On a UNIX system, go to the subdirectory `examples/`*platform*`/`*libformat* that matches your particular platform of operating system and compiler. In that directory, you will find a file named `Makefile`. Execute one of the examples, for instance `lpex1.c` , by trying this sequence of commands:

```
make lpex1

lpex1 -r
```

This example takes one argument, either `-r` , `-c` , or `-n` .

If your interest is in running one of the C++ examples, try this sequence of commands to execute a comparable example in C++:

```
make ilolpex1

ilolpex1 -r
```

If your interest is in running one of the Java examples, try this sequence of commands:

```
make LPex1.class

java -Djava.library.path=../../../bin/platform: \

     -classpath ../../../lib/cplex.jar: LPex1 -r
```

where platform represents the *machine* and *library format*

Any of these examples, whether C, C++, or Java, should return an optimal objective function value of 202.5.

Verifying set-up on Windows

On a Windows machine, you can follow a similar process using the facilities of your compiler interface to compile and then run any of the examples. A project file for each example is provided, in a format for Microsoft Visual Studio.

To run the examples on Windows, either you must copy the CPLEX DLL file to the directory or folder containing the examples, or you must make sure that the location of the DLL file is part of your Windows path, as explained in "Setting up CPLEX on Windows" on page 1.

In case of errors

If an error occurs during the steps to `make` or to compile, then check that you are able to access the compiler and the necessary linker/loader files and system libraries. If an error occurs on the next step, when executing the program created by `make`, then the nature of the error message will guide your actions. For Windows users, if the program has trouble locating `cplex XXX .dll` or

ILOG.CPLEX.dll, make sure the DLL file is stored either in the current directory or in a directory listed in your PATH environment variable.

The UNIX Makefile, or Windows project file, contains useful information regarding recommended flags and other settings for compilation and linking.

Compiling and linking your own applications

The source files for the examples and the makefiles provide guidance for how your own application can call CPLEX. The following topics give more specific information about the necessary header files for compilation, and how to link CPLEX and Concert Technology libraries into your application.

- Chapter 4, “Concert Technology tutorial for C++ users,” on page 49 contains information and platform-specific instructions for compiling and linking the Concert Technology Library, for C++ users.
- Chapter 5, “Concert Technology tutorial for Java users,” on page 65 contains information and platform-specific instructions for compiling and linking the Concert Technology Library, for Java users.
- Chapter 6, “Concert Technology tutorial for .NET users,” on page 75 offers an example of a C#.NET application.
- Chapter 7, “Callable Library tutorial,” on page 83 contains information and platform-specific instructions for compiling and linking the Callable Library.
- Chapter 8, “Python tutorial,” on page 97 contains information about using conventional Python utilities, such as disutils, and instructions for launching an interactive Python session.

Chapter 2. Solving an LP with CPLEX

This example solves an LP model to contrast CPLEX components.

Overview

This example shows ways available in CPLEX to solve a linear programming problem.

To help you learn which IBM ILOG CPLEX component best meets your needs, this chapter briefly demonstrates how to create and solve an LP model. It shows you at a glance the Interactive Optimizer and the application programming interfaces (APIs) to CPLEX. Full details of writing a practical program are in the topics containing the tutorials.

Problem statement

This linear programming model in a standard formulation can be solved in each of the components of CPLEX.

The problem to be solved is:

Maximize	$x_1 + 2x_2 + 3x_3$
subject to	
	$-x_1 + x_2 + x_3 \leq 20$
	$x_1 - 3x_2 + x_3 \leq 30$
with these bounds	
	$0 \leq x_1 \leq 40$
	$0 \leq x_2 \leq infinity$
	$0 \leq x_3 \leq infinity$

Using the Interactive Optimizer

The Interactive Optimizer solves the model in this way.

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

```
Welcome to IBM(R) ILOG(R) CPLEX(R) Interactive Optimizer 12.6.0.0
  with Simplex, Mixed Integer & Barrier Optimizers
Copyright IBM Corp. 1988, 2013 All Rights Reserved.

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

CPLEX> enter example
Enter new problem ['end' on a separate line terminates]:
maximize  x1 + 2 x2 + 3 x3
subject to -x1 +  x2 + x3 <= 20
            x1 - 3 x2 + x3 <=30
bounds
0 <= x1 <= 40
```

© Copyright IBM Corp. 1987, 2013

```
0 <= x2
0 <= x3
end
CPLEX> optimize
Tried aggregator 1 time.
No LP presolve or aggregator reductions.
Presolve time =    0.00 sec. (0.00 ticks)

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

Dual simplex - Optimal:  Objective =   2.0250000000e+002
Solution time =    0.01 sec.  Iterations = 2 (1)
Deterministic time = 0.00 ticks  (3.38 ticks/sec)

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

Using Concert Technology in C++

This C++ application solves the model in this way.

Here is a C++ application using CPLEX in Concert Technology to solve the example. An expanded form of this example is discussed in detail in Chapter 4, “Concert Technology tutorial for C++ users,” on page 49.

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

int
main (int argc, char **argv)
{
   IloEnv env;
   try {
      IloModel model(env);
      IloNumVarArray vars(env);
      vars.add(IloNumVar(env, 0.0, 40.0));
      vars.add(IloNumVar(env));
      vars.add(IloNumVar(env));
        model.add(IloMaximize(env, vars[0] + 2 * vars[1] + 3 * vars[2]));
      model.add( - vars[0] +     vars[1] + vars[2] <= 20);
      model.add(   vars[0] - 3 * vars[1] + vars[2] <= 30);

      IloCplex cplex(model);
      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, vars);
      env.out() << "Values = " << vals << endl;
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }
```

```
   env.end();

   return 0;
}
```

Using Concert Technology in Java

This Java application solves the model in this way.

Here is a Java application using CPLEX with Concert Technology to solve the example. An expanded form of this example is discussed in detail in Chapter 5, “Concert Technology tutorial for Java users,” on page 65.

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

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

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

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

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

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

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

Using Concert Technology in .NET

This C#.NET application solves the model in this way..

Here is a C#NET application using Concert Technology with CPLEX to solve the example. A tutorial offering an expanded version of this application is available in Chapter 6, “Concert Technology tutorial for .NET users,” on page 75.

```
using ILOG.Concert;
using ILOG.CPLEX;
public class Example {
   public static void Main(string[] args) {
```

```
      try {
         Cplex cplex = new Cplex();
         double[]  lb = {0.0, 0.0, 0.0};
         double[]  ub = {40.0, System.Double.MaxValue, System.Double.MaxValue};
         INumVar[] x  = cplex.NumVarArray(3, lb, ub);
         var[0] = x;
         double[] objvals = {1.0, 2.0, 3.0};
         cplex.Add(cplex.Maximize(cplex.ScalProd(x, objvals)));
         rng[0] = new IRange[2];
         rng[0][0] = cplex.AddRange(-System.Double.MaxValue, 20.0);
         rng[0][1] = cplex.AddRange(-System.Double.MaxValue, 30.0);
         rng[0][0].Expr = cplex.Sum(cplex.Prod(-1.0, x[0]),
                                    cplex.Prod( 1.0, x[1]),
                                    cplex.Prod( 1.0, x[2]));
         rng[0][1].Expr = cplex.Sum(cplex.Prod( 1.0, x[0]),
                                    cplex.Prod(-3.0, x[1]),
                                    cplex.Prod( 1.0, x[2]));
         x[0].Name = "x1";
         x[1].Name = "x2";
         x[2].Name = "x3";
         rng[0][0].Name = "c1";
         rng[0][0].Name = "c2";
         cplex.ExportModel("example.lp");
         if ( cplex.Solve() ) {
            double[] x     = cplex.GetValues(var[0]);
            double[] dj    = cplex.GetReducedCosts(var[0]);
            double[] pi    = cplex.GetDuals(rng[0]);
            double[] slack = cplex.GetSlacks(rng[0]);
            cplex.Output().WriteLine("Solution status = " + cplex.GetStatus());
            cplex.Output().WriteLine("Solution value  = " + cplex.ObjValue);
            int nvars = x.Length;
            for (int j = 0; j < nvars; ++j) {
               cplex.Output().WriteLine("Variable   " + j +
                                        ": Value = " + x[j] +
                                        " Reduced cost = " + dj[j]);
            }
            int ncons = slack.Length;
            for (int i = 0; i < ncons; ++i) {
               cplex.Output().WriteLine("Constraint " + i +
                                        ": Slack = " + slack[i] +
                                        " Pi = " + pi[i]);
            }
         }
         cplex.End();
      }
      catch (ILOG.Concert.Exception e) {
         System.Console.WriteLine("Concert exception '" + e + "' caught");
      }
   }
}
```

Using the Callable Library

This C application solves the model in this way..

Here is a C application using the CPLEX Callable Library to solve the example. An expanded form of this example is discussed in detail in Chapter 7, “Callable Library tutorial,” on page 83.

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

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

```
int
main (int argc, char **argv)
{
   int       status = 0;
   CPXENVptr env = NULL;
   CPXLPptr  lp  = NULL;

   double   obj[NUMCOLS];
   double   lb[NUMCOLS];
   double   ub[NUMCOLS];
   double   x[NUMCOLS];
   int      rmatbeg[NUMROWS];
   int      rmatind[NUMNZ];
   double   rmatval[NUMNZ];
   double   rhs[NUMROWS];
   char     sense[NUMROWS];

   int      solstat;
   double   objval;

   env = CPXopenCPLEX (&status);
   if ( env == NULL ) {
      char  errmsg[1024];
      fprintf (stderr, "Could not open CPLEX environment.\n");
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, "%s", errmsg);
      goto TERMINATE;
   }
   lp = CPXcreateprob (env, &status, "lpex1");
   if ( lp == NULL ) {
      fprintf (stderr, "Failed to create LP.\n");
      goto TERMINATE;
   }

   CPXchgobjsen (env, lp, CPX_MAX);

       obj[0] = 1.0;      obj[1] = 2.0;           obj[2] = 3.0;
        lb[0] = 0.0;       lb[1] = 0.0;           lb[2]  = 0.0;
        ub[0] = 40.0;      ub[1] = CPX_INFBOUND;  ub[2]  = CPX_INFBOUND;

   status = CPXnewcols (env, lp, NUMCOLS, obj, lb, ub, NULL, NULL);
   if ( status ) {
      fprintf (stderr, "Failed to populate problem.\n");
      goto TERMINATE;
   }

   rmatbeg[0] = 0;
   rmatind[0] = 0;     rmatind[1] = 1;    rmatind[2] = 2;  sense[0] = 'L';
   rmatval[0] = -1.0;  rmatval[1] = 1.0;  rmatval[2] = 1.0;  rhs[0] = 20.0;

   rmatbeg[1] = 3;
   rmatind[3] = 0;    rmatind[4] = 1;    rmatind[5] = 2;     sense[1] = 'L';
   rmatval[3] = 1.0;  rmatval[4] = -3.0; rmatval[5] = 1.0;   rhs[1]   = 30.0;

   status = CPXaddrows (env, lp, 0, NUMROWS, NUMNZ, rhs, sense, rmatbeg,
                        rmatind, rmatval, NULL, NULL);
   if ( status ) {
      fprintf (stderr, "Failed to populate problem.\n");
      goto TERMINATE;
   }

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

```
   status = CPXsolution (env, lp, &solstat, &objval, x, NULL, NULL, NULL);
   if ( status ) {
      fprintf (stderr, "Failed to obtain solution.\n");
      goto TERMINATE;
   }
   printf ("\nSolution status = %d\n", solstat);
   printf ("Solution value  = %f\n", objval);
   printf ("Solution        = [%f, %f, %f]\n\n", x[0], x[1], x[2]);
TERMINATE:

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

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

Using the Python API

This Python application solves the model in this way.

Here is a Python application using CPLEX to solve the example.

```
execfile("cplexpypath.py")

import cplex
from cplex.exceptions import CplexError
import sys

# data common to all populateby functions
my_obj      = [1.0, 2.0, 3.0]
my_ub       = [40.0, cplex.infinity, cplex.infinity]
my_colnames = ["x1", "x2", "x3"]
my_rhs      = [20.0, 30.0]
my_rownames = ["c1", "c2"]
my_sense    = "LL"


def populatebyrow(prob):
    prob.objective.set_sense(prob.objective.sense.maximize)

    # since lower bounds are all 0.0 (the default), lb is omitted here
    prob.variables.add(obj = my_obj, ub = my_ub, names = my_colnames)

    # can query variables like the following bounds and names:

    # lbs is a list of all the lower bounds
    lbs = prob.variables.get_lower_bounds()

    # ub1 is just the first lower bound
    ub1 = prob.variables.get_upper_bounds(0)
```

```
    # names is ["x1", "x3"]
    names = prob.variables.get_names([0, 2])

    rows = [[[0,"x2","x3"],[-1.0, 1.0,1.0]],
            [["x1",1,2],[ 1.0,-3.0,1.0]]]

    prob.linear_constraints.add(lin_expr = rows, senses = my_sense,
                                rhs = my_rhs, names = my_rownames)

    # because there are two arguments, they are taken to specify a range
    # thus, cols is the entire constraint matrix as a list of column vectors
    cols = prob.variables.get_cols("x1", "x3")


def populatebycolumn(prob):
    prob.objective.set_sense(prob.objective.sense.maximize)

    prob.linear_constraints.add(rhs = my_rhs, senses = my_sense,
                                names = my_rownames)

    c = [[[0,1],[-1.0, 1.0]],
         [["c1",1],[ 1.0,-3.0]],
         [[0,"c2"],[ 1.0, 1.0]]]

    prob.variables.add(obj = my_obj, ub = my_ub, names = my_colnames,
                       columns = c)

def populatebynonzero(prob):
    prob.objective.set_sense(prob.objective.sense.maximize)

    prob.linear_constraints.add(rhs = my_rhs, senses = my_sense,
                                names = my_rownames)
    prob.variables.add(obj = my_obj, ub = my_ub, names = my_colnames)

    rows = [0,0,0,1,1,1]
    cols = [0,1,2,0,1,2]
    vals = [-1.0,1.0,1.0,1.0,-3.0,1.0]

    prob.linear_constraints.set_coefficients(zip(rows, cols, vals))
    # can also change one coefficient at a time

    # prob.linear_constraints.set_coefficients(1,1,-3.0)
    # or pass in a list of triples
    # prob.linear_constraints.set_coefficients([(0,1,1.0), (1,1,-3.0)])


def lpex1(pop_method):
    try:
        my_prob = cplex.Cplex()

        if pop_method == "r":
            handle = populatebyrow(my_prob)
        if pop_method == "c":
            handle = populatebycolumn(my_prob)
        if pop_method == "n":
            handle = populatebynonzero(my_prob)

        my_prob.solve()
    except CplexError, exc:
        print exc
        return

    numrows = my_prob.linear_constraints.get_num()
    numcols = my_prob.variables.get_num()

    print
```

```
    # solution.get_status() returns an integer code
    print "Solution status = " , my_prob.solution.get_status(), ":",
    # the following line prints the corresponding string
    print my_prob.solution.status[my_prob.solution.get_status()]
    print "Solution value  = ", my_prob.solution.get_objective_value()
    slack = my_prob.solution.get_linear_slacks()
    pi    = my_prob.solution.get_dual_values()
    x     = my_prob.solution.get_values()
    dj    = my_prob.solution.get_reduced_costs()
    for i in range(numrows):
        print "Row %d:  Slack = %10f  Pi = %10f" % (i, slack[i], pi[i])
    for j in range(numcols):
        print "Column %d:  Value = %10f Reduced cost = %10f" % (j, x[j], dj[j])

    my_prob.write("lpex1.lp")

if __name__ == "__main__":
    if len(sys.argv) != 2 or sys.argv[1] not in  ["-r", "-c", "-n"]:
        print "Usage: lpex1.py -X"
        print "   where X is one of the following options:"
        print "      r          generate problem by row"
        print "      c          generate problem by column"
        print "      n          generate problem by nonzero"
        print " Exiting..."
        sys.exit(-1)
    lpex1(sys.argv[1][1])
else:
    prompt = """Enter the letter indicating how the problem data should be populated:
    r : populate by rows
    c : populate by columns
    n : populate by nonzeros\n ? > """
    r = 'r'
    c = 'c'
    n = 'n'
    lpex1(input(prompt))
```

Chapter 3. Interactive Optimizer tutorial

The major features of the CPLEX Interactive Optimizer are introduced in this tutorial.

Starting CPLEX

The start command launches the Interactive Optimizer.

Procedure

To start the CPLEX Interactive Optimizer, at your operating system prompt type the command:

```
cplex
```

A message similar to the following one appears on the screen:

```
Welcome to CPLEX Interactive Optimizer 12.6.0
  with Simplex, Mixed Integer, & Barrier Optimizers
Copyright (c) IBM 1997-2013
CPLEX is a registered trademark of IBM(r)

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


CPLEX>
```

Results

The last line, `CPLEX>` , is the prompt, showing that the product is running and is ready to accept one of the available CPLEX commands. Use the `help` command to see a list of these commands.

Using help

The help command invokes help in the Interactive Optimizer.

About this task

CPLEX accepts commands in several different formats. You can type either the full command name, or any shortened form that uniquely identifies that name.

Procedure

For example, enter `help` after the `CPLEX>` prompt, as shown:

```
CPLEX> help
```

You will see a list of the CPLEX commands on the screen.
Since all commands start with a unique letter, you could also enter just the single letter `h` .

```
CPLEX> h
```

© Copyright IBM Corp. 1987, 2013

CPLEX does not distinguish between upper- and lower-case letters, so you could enter `h` , `H` , `help` , or `HELP`. All of these variations invoke the `help` command. The same rules apply to all CPLEX commands. You need to type only enough letters of the command to distinguish it from all other commands, and it does not matter whether you type upper- or lower-case letters. This manual uses lower-case letters.

Results

After you type the `help` command, a list of available commands with their descriptions appears on the screen, like this:

```
add         add constraints to the problem
baropt      solve using barrier algorithm
change      change the problem
conflict    refine a conflict for an infeasible problem
display     display problem, solution, or parameter settings
enter       enter a new problem
feasopt     find relaxation to an infeasible problem
help        provide information on CPLEX commands
mipopt      solve a mixed integer program
netopt      solve the problem using network method
optimize    solve the problem
populate    get additional solutions for a mixed integer program
primopt     solve using the primal method
quit        leave CPLEX
read        read problem or advanced start information from a file
set         set parameters
tranopt     solve using the dual method
tune        try a variety of parameter settings
write       write problem or solution information to a file
xecute      execute a command from the operating system

Enter enough characters to uniquely identify commands & options. Commands can beentered partially (C
```

To find out more about a specific command, type `help` followed by the name of that command. For example, to learn more about the `primopt` command type:

```
help primopt
```

Typing the full name is unnecessary. Alternatively, you can try:

```
h p
```

The following message appears to tell you more about the use and syntax of the `primopt` command:

```
The PRIMOPT command solves the current problem using
a primal simplex method or crosses over to a basic solution
if a barrier solution exists.

Syntax:  PRIMOPT

A problem must exist in memory (from using either the
ENTER or READ command) in order to use the PRIMOPT
command.

Sensitivity information (dual price and reduced-cost
information) as well as other detailed information about
the solution can be viewed using the DISPLAY command,
after a solution is generated.
```

The syntax for the `help` command is:

`help` *command name*

Entering a problem

The Interactive Optimizer offers a variety of ways to enter a problem.

Overview

The Interactive Optimizer supports manual entry of data for a problem.

Most users with larger problems enter problems by reading data from formatted files. That practice is explained in “Reading problem files” on page 36. For now, you will enter a smaller problem from the keyboard by using the `enter` command. The process is outlined step-by-step in the following topics.

Entering the example

This example appears throughout this tutorial about the Interactive Optimizer.

As an example, this manual uses the following problem:

Maximize	$x_1 + 2x_2 + 3x_3$
subject to	
	$-x_1 + x_2 + x_3 \leq 20$
	$x_1 - 3x_2 + x_3 \leq 30$
with these bounds	
	$0 \leq x_1 \leq 40$
	$0 \leq x_2 \leq infinity$
	$0 \leq x_3 \leq infinity$

This problem has three variables (x1, x2, and x3) and two less-than-or-equal-to constraints.

The `enter` command is used to enter a new problem from the keyboard. The procedure is almost as simple as typing the problem on a page. At the `CPLEX>` prompt type:

```
enter
```

A prompt appears on the screen asking you to give a name to the problem that you are about to enter.

Naming a problem

The problem name may be anything that is allowed as a file name in your operating system. If you decide that you do not want to enter a new problem, just press the `<return>` key without typing anything. The `CPLEX>` prompt will reappear without causing any action. The same can be done at any `CPLEX>` prompt. If you do not want to complete the command, simply press the `<return>` key. For now, type in the name `example` at the prompt.

```
Enter name for problem: example
```

The following message appears:

```
Enter new problem ['end' on a separate line terminates]:
```

and the cursor is positioned on a blank line below it where you can enter the new problem.

You can also type the problem name directly after the `enter` command and avoid the intermediate prompt.

Summary

The syntax for entering a problem is:

```
enter problem name
```

Using the LP format

The Interactive Optimizer supports LP file format to enter a problem.

Entering a new problem is basically like typing it on a page, but there are a few rules to remember. These rules conform to the CPLEX LP file format and are documented in the *CPLEX File Formats Reference Manual*. LP format appears throughout this tutorial.

The problem should be entered in the following order:

1. "Objective function"
2. "Constraints"
3. "Bounds" on page 23

Objective function

Before entering the objective function, you must state whether the problem is a minimization or maximization. For this example, you type:

```
maximize
x1 + 2x2 + 3x3
```

You may type `minimize` or `maximize` on the same line as the objective function, but you must separate them by at least one space.

Variable names

In the example, the variables are named simply `x1` , `x2` , `x3` , but you can give your variables more meaningful names such as `cars` or `gallons` . The only limitations on variable names in LP format are that the names must be no more than 255 characters long and use only the alphanumeric characters (a-z, A-Z, 0-9) and certain symbols: ! " # $ % & ( ) , . ; ? @ _ ' ' { } ~. Any line with more than 510 characters is truncated.

A variable name cannot begin with a number or a period, and there is one character combination that cannot be used: the letter `e` or `E` alone or followed by a number or another `e` , since this notation is reserved for exponents. Thus, a variable cannot be named `e24` nor `e9cats` nor `eels` nor any other name with this pattern. This restriction applies only to problems entered in LP format.

Constraints

After you have entered the objective function, you can move on to the constraints. However, before you start entering the constraints, you must indicate that the subsequent lines are constraints by typing:

```
subject to
```

or

```
st
```

These terms can be placed alone on a line or on the same line as the first constraint if separated by at least one space. Now you can type in the constraints in the following way:

```
st
-x1 + x2 + x3 <= 20
x1 - 3x2 + x3 <= 30
```

Constraint names

In this simple example, it is easy to keep track of the small number of constraints, but for many problems, it may be advantageous to name constraints so that they are easier to identify. You can do so in CPLEX by typing a constraint name and a colon before the actual constraint. If you do not give the constraints explicit names, CPLEX will give them the default names `c1, c2, . . . , cn` . In the example, if you want to call the constraints `time` and `labor` , for example, enter the constraints like this:

```
st
time:  -x1 + x2 + x3 <= 20
labor:  x1 - 3x2 + x3 <= 30
```

Constraint names are subject to the same guidelines as variable names. They must have no more than 255 characters, consist of only allowed characters, and not begin with a number, a period, or the letter `e` followed by a positive or negative number or another `e` .

Objective function names

The objective function can be named in the same manner as constraints. The default name for the objective function is `obj`. CPLEX assigns this name if no other is entered.

Bounds

Finally, you must enter the lower and upper bounds on the variables. If no bounds are specified, CPLEX will automatically set the lower bound to 0 and the upper bound to +∞. You must explicitly enter bounds only when the bounds differ from the default values. In our example, the lower bound on `x1` is 0, which is the same as the default. The upper bound 40, however, is not the default, so you must enter it explicitly. You must type `bounds` on a separate line before you enter the bound information:

```
bounds
x1 <= 40
```

Since the bounds on `x2` and `x3` are the same as the default bounds, there is no need to enter them. You have finished entering the problem, so to indicate that the problem is complete, type:

```
end
```

on the last line.

The `CPLEX>` prompt returns, indicating that you can again enter a CPLEX command.

Summary

Entering a problem in CPLEX is straightforward, provided that you observe a few simple rules:

- The terms maximize or minimize must precede the objective function; the term subject to must precede the constraints section; both must be separated from the beginning of each section by at least one space.
- The word bounds must be alone on a line preceding the bounds section.
- On the final line of the problem, end must appear.

Entering data

Entering data from the keyboard entails these special considerations.

You can use the <return> key to split long constraints, and CPLEX still interprets the multiple lines as a single constraint. When you split a constraint in this way, do not press <return> in the middle of a variable name or coefficient. The following sequence is acceptable:

```
time: -x1 + x2 + <return>
x3 <= 20  <return>
labor:  x1 - 3x2 + x3 <= 30  <return>
```

The entry below, however, is incorrect since the <return> key splits a variable name.

```
time: -x1 + x2 + x <return>
3 <= 20  <return>
labor:  x1 - 3x2 + x3 <= 30  <return>
```

If you type a line that CPLEX cannot interpret, a message indicating the problem will appear, and the entire constraint or objective function will be ignored. You must then re-enter the constraint or objective function.

The final point to remember when you are entering a problem is that after you have pressed <return> , you can no longer directly edit the characters that precede the <return>. As long as you have not pressed the <return> key, you can use the <backspace> key to go back and change what you typed on that line. After you press <return>, you must use the change command to modify the problem. The change command is documented in "Changing a problem" on page 41.

Displaying a problem

The Interactive Optimizer displays problems according to your commands.

Verifying a problem with the display command

The display command in the Interactive Optimizer supports a variety of options.

Now that you have entered a problem using CPLEX, you must verify that the problem was entered correctly. To do so, use the display command. At the CPLEX> prompt type:

```
display
```

A list of the items that can be displayed then appears. Some of the options display parts of the problem description, while others display parts of the problem solution. Options about the problem solution are not available until after the problem has been solved. The list looks like this:

```
Display Options:

auxiliary  display auxiliary information used during optimization
conflict    display conflict that demonstrates model infeasibility
problem     display problem characteristics
sensitivity display sensitivity analysis
settings    display parameter settings
solution    display existing solution

Display what:
```

If you type `problem` in reply to that prompt, that option will list a set of problem characteristics, like this:

```
Display Problem Options:

all             display entire problem
binaries        display binary variables
bounds          display a set of bounds
constraints     display a set of constraints or node supply/demand values
generals        display general integer variables
histogram       display a histogram of row or column counts
indicators      display a set of indicator constraints
integers        display integer variables
names           display names of variables or constraints
qpvariables     display quadratic variables
qconstraints    display quadratic constraints
semi-continuous display semi-continuous and semi-integer variables
sos             display special ordered sets
stats           display problem statistics
variable        display a column of the constraint matrix

Display which problem characteristic:
```

Enter the option `all` to display the entire problem.

```
Maximize
 obj: x1 + 2 x2 + 3 x3
Subject To
 c1: - x1 +   x2 +   x3 <= 20
 c2:  x1 - 3 x2 +   x3 <= 30
Bounds
 0 <= x1 <= 40
All other variables are >= 0.
```

The default names `obj` , `c1` , `c2` , are provided by CPLEX.

If that is what you want, you are ready to solve the problem. If there is a mistake, you must use the `change` command to modify the problem. The `change` command is documented in “Changing a problem” on page 41.

Summary

Display problem characteristics by entering the command:

```
display problem
```

Displaying problem statistics

These options for displaying information in the Interactive Optimizer support large problems.

When the problem is as small as our example, it is easy to display it on the screen; however, many real problems are far too large to display. For these problems, the

`stats` option of the `display problem` command is helpful. When you select `stats` , information about the attributes of the problem appears, but not the entire problem itself. These attributes include:

- the number and type of constraints
- variables
- nonzero constraint coefficients

Try this feature by typing:

```
display problem stats
```

For our example, the following information appears:

```
Problem name: example
Variables            :       3  [Nneg: 2,  Box: 1]
Objective nonzeros   :       3
Linear constraints   :       2  [Less: 2]
  Nonzeros           :       6
  RHS nonzeros       :       2
```

This information tells us that in the example there are two constraints, three variables, and six nonzero constraint coefficients. The two constraints are both of the type less-than-or-equal-to. Two of the three variables have the default nonnegativity bounds ($0 \le x \le +\infty$) and one is restricted to a certain range (a box variable). In addition to a constraint matrix nonzero count, there is a count of nonzero coefficients in the objective function and on the righthand side. Such statistics can help to identify errors in a problem without displaying it in its entirety. The command `display problem stats` shows this additional information like this:

```
Variables            : Min LB: 0.000000         Max UB: 40.00000
Objective nonzeros   : Min   : 1.000000         Max   : 3.000000
Linear constraints   :
  Nonzeros           : Min   : 1.000000         Max   : 3.000000
  RHS nonzeros       : Min   : 20.00000         Max   : 30.00000
```

Another way to avoid displaying an entire problem is to display a specific part of it by using one of the following three options of the `display problem` command:

- `names` , documented in "Displaying variable or constraint names" on page 27, can be used to display a specified set of variable or constraint names;
- `constraints` , documented in "Displaying constraints" on page 28, can be used to display a specified set of constraints;
- `bounds` , documented in "Displaying bounds" on page 28, can be used to display a specified set of bounds.

Specifying item ranges

These conventions in notation display ranges of items in the Interactive Optimizer.

For some options of the `display` command, you must specify the item or range of items you want to see. Whenever input defining a range of items is required, CPLEX expects two indices separated by a hyphen (the range character `-`). The indices can be names or matrix index numbers. You simply enter the starting name (or index number), a hyphen (–), and finally the ending name (or index number). CPLEX automatically sets the default upper and lower limits defining any range to be the highest and lowest possible values. Therefore, you have the option of leaving out either the upper or lower name (or index number) on either side of the hyphen. To see every possible item, you would simply enter –.

Another way to specify a range of items is to use a wildcard. CPLEX accepts these wildcards in place of the hyphen to specify a range of items:

- question mark (?) for a single character;
- asterisk (*) for zero or more characters.

For example, to specify all items, you could enter * (instead of -) if you want.

The sequence of characters c1? matches the name of every constraint in the range from c10 to c19 , for example.

Displaying variable or constraint names

These options display names of variables or constraints in the Interactive Optimizer.

You can display a variable name by using the display command with the options problem names variables. If you do not enter the word variables, CPLEX prompts you to specify whether you wish to see a constraint or variable name.

Type the following command:

```
display problem names variables
```

In response, CPLEX prompts you to specify a set of variable names to be displayed, like this:

```
Display which variable name(s):
```

Specify these variables by entering the names of the variables or the numbers corresponding to the columns of those variables. A single number can be used or a range such as 1-2 . All of the names can be displayed if you type a hyphen (the character -). Try this by entering a hyphen at the prompt and pressing the <return> key.

```
Display which variable name(s): -
```

You could also use a wildcard to display variable names, like this:

```
Display which variable name(s): *
```

In the example, there are three variables with default names. CPLEX displays these three names:

```
x1  x2  x3
```

If you want to see only the second and third names, you could either enter the range as 2-3 or specify everything following the second variable with 2-. Try this technique:

```
display problem names variables
Display which variable name(s): 2-
x2  x3
```

If you enter a number without a hyphen, you will see a single variable name:

```
display problem names variables
Display which variable name(s): 2
x2
```

Summary

- You can use a wildcard in the display command to specify a range of items.
- You can display variable names by entering the command:

```
display problem names variables
```

- You can display constraint names by entering the command:

```
display problem names constraints
```

Ordering variables

The Interactive Optimizer respects this internal order among variables.

In the example problem there is a direct correlation between the variable names and their numbers (`x1` is variable 1, `x2` is variable 2, etc.); that is not always the case. The internal ordering of the variables is based on their order of occurrence when the problem is entered. For example, if `x2` had not appeared in the objective function, then the order of the variables would be `x1` , `x3` , `x2` .

You can see the internal ordering by using the hyphen when you specify the range for the `variables` option. The variables are displayed in the order corresponding to their internal ordering.

All of the options of the `display` command can be entered directly after the word `display` to eliminate intermediate steps. The following command is correct, for example:

```
display problem names variables 2-3
```

Displaying constraints

These options display constraints in the Interactive Optimizer.

To view a single constraint within the matrix, use the command and the constraint number. For example, type the following command:

```
display problem constraints 2
```

The second constraint appears:

```
c2: x1 - 3 x2 + x3 <= 30
```

You can also use a wildcard to display a range of constraints, like this:

```
display problem constraints *
```

Displaying the objective function

These options display an objective function in the Interactive Optimizer.

When you want to display only the objective function, you must enter its name (`obj` by default) or an index number of 0.

```
display problem constraints
Display which constraint name(s): 0
Maximize
  obj: x1 + 2 x2 + 3 x3
```

Displaying bounds

These options display bounds of a problem in the Interactive Optimizer.

To see only the bounds for the problem, type the following command (don't forget the hyphen or wildcard):

```
display problem bounds -
```

or, try a wildcard, like this:

```
display problem bounds *
```

The result is:

```
0 <= x1 <= 40
All other variables are >= 0.
```

Summary

The general syntax of the `display` command is:

```
display option [option2] identifier - [identifier2]
```

Displaying a histogram of nonzero counts

These options display a summary of nonzero rows and columns in the Interactive Optimizer.

For large models, it can sometimes be helpful to see summaries of nonzero counts of the columns or rows of the constraint matrix. This kind of display is known as a *histogram*. There are two commands for displaying histograms: one for columns, one for rows.

```
display problem histogram c
```

```
display problem histogram r
```

For the small example in this tutorial, the column histogram looks like this:

```
Column counts (excluding fixed variables):

    Nonzero Count:   2
Number of Columns:   3
```

It tells you that there are three columns each having two nonzeroes, and no other columns. Similarly, the row histogram of the same small problem looks like this:

```
Row counts (excluding fixed variables):

 Nonzero Count:   3
Number of Rows:   2
```

It tells you that there are two rows with three nonzeroes in each of them.

Of course, in a more complex model, there would usually be a wider variety of nonzero counts than those histograms show. Here is an example in which there are sixteen columns where only one row is nonzero, 756 columns where two rows are nonzero, and so forth.

```
Column counts (excluding fixed variables):
Nonzero Count:      1   2     3    4     5    6  15  16
Number of Columns: 16 756  1054  547   267  113   2   1
```

If there has been an error during entry of the problem, perhaps a constraint coefficient having been omitted by mistake, for example, summaries like these, of a model where the structure of the constraint matrix is known, may help you find the source of the error.

Solving a problem

These commands solve a problem in the Interactive Optimizer.

Where you are

To solve a model and display its solution in the Interactive Optimizer, use commands and options.

If you have been following this tutorial step by step, the problem is now correctly entered, and you can now use CPLEX to solve it. This tutorial continues with topics about solving the problem and displaying solution information. Each of these topics introduces commands and options.

Solving the example

While solving a problem, the Interactive Optimizer executes the `optimize` command.

The `optimize` command tells CPLEX to solve the LP problem. CPLEX uses the dual simplex optimizer, unless another method has been specified by setting the `LPMETHOD` parameter (explained more fully in the *CPLEX User's Manual*).

Entering the optimize command

At the `CPLEX>` prompt, type the command:

```
optimize
```

Preprocessing

First, CPLEX tries to simplify or reduce the problem using its presolver and aggregator. If any reductions are made, a message will appear. However, in our small example, no reductions are possible.

Monitoring the iteration log

Next, an iteration log appears on the screen. CPLEX reports its progress as it solves the problem. The solution process involves two stages:

- during Phase I, CPLEX searches for a feasible solution
- in Phase II, CPLEX searches for the optimal feasible solution.

The iteration log periodically displays the current iteration number and either the current scaled infeasibility during Phase I, or the objective function value during Phase II. After the optimal solution has been found, the objective function value, solution time, and iteration count (total, with Phase I in parentheses) are displayed. This information can be useful for monitoring the rate of progress.

The iteration log display can be modified by the command `set simplex display` to display differing amounts of data while the problem is being solved.

Reporting the solution

After it finds the optimal solution, CPLEX reports:

- the objective function value
- the problem solution time in seconds
- the total iteration count
- the Phase I iteration count (in parentheses)

Optimizing our example problem produces a report like the following one (although the solution times vary with each computer):

```
Tried aggregator 1 time.
No LP presolve or aggregator reductions.
Presolve Time = 0.00 sec. (0.00 ticks)

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

Dual simplex - Optimal:  Objective =    2.0250000000e+02
Solution Time =    0.00 sec.  Iterations = 2 (1)
Deterministic time = 0.00 ticks  (3.38 ticks/sec)

CPLEX>
```

In our example, CPLEX finds an optimal solution with an objective value of 202.5 in two iterations. For this simple problem, 1 Phase I iteration was required.

Summary

To solve an LP problem, use the command:

```
optimize
```

Solution options

After solving a problem, the Interactive Optimizer supports these additional options.

Here are some of the basic options in solving linear programming problems. Although the example in this tutorial does not make use of these options, you will find them useful when handling larger, more realistic problems.

- "Filing iteration logs";
- "Re-solving";
- "Using alternative optimizers" on page 32;
- "Interrupting the optimization" on page 32.

For detailed information about performance options, refer to the *CPLEX User's Manual*.

Filing iteration logs

Every time CPLEX solves a problem, much of the information appearing on the screen is also directed into a log file. This file is automatically created by CPLEX with the name `cplex.log` . If there is an existing `cplex.log` file in the directory where CPLEX is launched, CPLEX will append the current session data to the existing file. If you want to keep a unique log file of a problem session, you can change the default name with the `set logfile` command. (See the *CPLEX User's Manual*.) The log file is written in standard ASCII format and can be edited with any text editor.

Re-solving

You may re-solve the problem by reissuing the `optimize` command. CPLEX restarts the solution process from the previous optimal basis, and thus requires zero iterations. If you do not wish to restart the problem from an advanced basis, use the `set advance` command to turn off the advanced start indicator.

Remember that a problem must be present in memory (entered via the `enter` command or read from a file) before you issue the `optimize` command.

Using alternative optimizers

In addition to the `optimize` command, CPLEX can use the primal simplex optimizer (`primopt` command), the dual simplex optimizer (`tranopt` command), the barrier optimizer (`baropt` command) and the network optimizer (`netopt` command). Many problems can be solved faster using these alternative optimizers, which are documented in more detail in the *CPLEX User's Manual*. If you want to solve a mixed integer programming problem, the `optimize` command is equivalent to the `mipopt` command.

Interrupting the optimization

Our short example was solved very quickly. However, larger problems, particularly mixed integer problems, can take much longer. Occasionally it may be useful to interrupt the optimization process. CPLEX allows such interruptions if you use `control-c` . (The `control` and `c` keys must be pressed simultaneously.) Optimization is interrupted, and CPLEX issues a message indicating that the process was stopped and displays progress information. If you issue another optimization command in the same session, CPLEX will resume optimization from where it was interrupted.

Displaying post-solution information

For post-solution information, the Interactive Optimizer supports these options.

After an optimal solution is found, CPLEX can provide many different kinds of information for you to view and analyze the results. You access this information by means of the `display` command and by certain `write` commands.

Information about the following features of a solution is available with the `display solution` command:

- objective function value;
- solution values;
- numerical quality of the solution;
- slack values;
- reduced costs;
- dual values (shadow prices);
- basic rows and columns.

For information about the `write` commands, see "Writing problem and solution files" on page 34. Sensitivity analysis can also be performed in analyzing results, as explained in "Performing sensitivity analysis" on page 33.

For example, to view the optimal value of each variable, enter the command:

```
display solution variables -
```

In response, the list of variable names with the solution value for each variable is displayed, like this:

```
Variable Name    Solution Value
x1                    40.000000
x2                    17.500000
x3                    42.500000
```

To view the slack values of each constraint, enter the command:

```
display solution slacks -
```

The resulting message indicates that for this problem the slack variables are all zero.

```
All slacks in the range 1-2 are 0.
```

To view the dual values (sometimes called shadow prices) for each constraint, enter the command:

```
display solution dual -
```

The list of constraint names with the solution value for each constraint appears, like this:

```
Constraint Name     Dual Price
c1                    2.750000
c2                    0.250000
```

Summary

Display solution characteristics by entering a command with the syntax:

```
display solution identifier
```

Performing sensitivity analysis

To perform sensitivity analysis, the Interactive Optimizer supports these options.

Sensitivity analysis of the objective function and righthand side provides meaningful insight about ways in which the optimal solution of a problem changes in response to small changes in these parts of the problem data.

Sensitivity analysis can be performed on the following features of a solution:

- objective function;
- righthand side values;
- bounds.

To view the sensitivity analysis of the objective function, enter the command:

```
display sensitivity obj -
```

You can also use a wildcard to query solution information, like this:

```
display sensitivity obj *
```

For our example, CPLEX displays the following ranges for sensitivity analysis of the objective function:

```
                               OBJ Sensitivity Ranges

Variable Name  Reduced Cost      Down   Current         Up
x1                   3.5000   -2.5000    1.0000   +infinity
x2                     zero   -5.0000    2.0000      3.0000
x3                     zero    2.0000    3.0000   +infinity
```

CPLEX displays each variable, its reduced cost and the range over which its objective function coefficient can vary without forcing a change in the optimal basis. The current value of each objective coefficient is also displayed for reference. Objective function sensitivity analysis is useful to analyze how sensitive the optimal solution is to the cost or profit associated with each variable.

Similarly, to view sensitivity analysis of the righthand side, type the command:

```
display sensitivity rhs -
```

For our example, CPLEX displays the following ranges for sensitivity analysis of the righthand side (RHS):

```
                                  RHS Sensitivity Ranges

Constraint Name Dual Price       Down   Current          Up
c1                  2.7500   -36.6667   20.0000   +infinity
c2                  0.2500  -140.0000   30.0000    100.0000
```

CPLEX displays each constraint, its dual price, and a range over which its righthand side coefficient can vary without changing the optimal basis. The current value of each RHS coefficient is also displayed for reference. Righthand side sensitivity information is useful for analyzing how sensitive the optimal solution and resource values are to the availability of those resources.

CPLEX can also display lower bound sensitivity ranges with the command

```
display sensitivity lb
```

and upper bound sensitivity with the command

```
display sensitivity ub
```

Summary

Display sensitivity analysis characteristics by entering a command with the syntax:

`display sensitivity` *`identifier`*

Writing problem and solution files

To write files, the Interactive Optimizer supports a variety of options.

Overview

The write command of the Interactive Optimizer records information in a file.

The problem or its solution can be saved by using the `write` command. This command writes the problem statement or a solution report to a file.

Selecting a write file format

The Interactive Optimizer writes files in a variety of formats, according to options you specify.

When you type the `write` command in the Interactive Optimizer, CPLEX displays a menu of options and prompts you for a file format, like this:

```
File type options:

alp     LP format problem with generic names and bound annotations
bas     INSERT format basis file
clp     Conflict file
dpe     Binary format for dual-perturbed problem
dua     MPS format of explicit dual of problem
emb     MPS format of (embedded) network
flt     Solution pool filters
lp      LP format problem file
min     DIMACS min-cost network-flow format of (embedded) network
mps     MPS format problem file
mst     MIP start file
```

```
net     CPLEX network format of (embedded) network
ord     Integer priority order file
ppe     Binary format for primal-perturbed problem
pre     Binary format for presolved problem
prm     Non-default parameter settings
rlp     LP format problem with generic names
rew     MPS format problem with generic names
sav     Binary matrix and basis file
sol     Solution file
```

```
File type:
```

- The BAS format is used for storing basis information and is introduced in "Writing basis files." See also "Reading basis files" on page 38.
- The LP format was discussed in "Using the LP format" on page 22. Using this format is explained in "Writing LP files" and "Reading LP files" on page 37.
- The MPS format is covered in "Reading MPS files" on page 38.

Note:

All these file formats are documented in more detail in the *CPLEX File Formats Reference Manual* .

Writing LP files

The command to write LP file format from the Interactive Optimizer prompts for options.

When you enter the `write` command. the following message appears:

```
Name of file to write:
```

Enter the problem name "example", and CPLEX will ask you to select a type from a list of options. For this example, choose LP. CPLEX displays a confirmation message, like this:

```
Problem written to file 'example'.
```

If you would like to save the file with a different name, you can simply use the `write` command with the new file name as an argument. Try this, using the name `example2` . This time, you can avoid intermediate prompts by specifying an LP problem type, like this:

```
write example2 lp
```

Another way of avoiding the prompt for a file format is by specifying the file type explicitly in the file name extension. Try the following example:

```
write example.lp
```

Using a file extension to indicate the file type is the recommended naming convention. This makes it easier to keep track of your problem and solution files.

When the file type is specified by the file name extension, CPLEX ignores subsequent file type information issued within the `write` command. For example, CPLEX responds to the following command by writing an LP format problem file:

```
write example.lp mps
```

Writing basis files

The Interactive Optimizer supports the option to write basis files.

Another optional file format is BAS. Unlike the LP and MPS formats, this format is not used to store a description of the problem statement. Rather, it is used to store information about the solution to a problem, information known as a *basis*. Even after changes are made to the problem, using a prior basis to start the optimization from an advanced basis can speed solution time considerably. A basis can be written only after a problem has been solved. Try this now with the following command:

```
write example.bas
```

In response, CPLEX displays a confirmation message, like this:

```
Basis written to file 'example.bas'.
```

Using path names

These special considerations with respect to path names apply in the Interactive Optimizer.

You can also include a full path name to specify to CPLEX on which drive and directory to save any file. The following `write` command is valid if the disk drive on your system contains a root directory named `problems`:

```
write /problems/example.lp
```

Summary

The general syntax for the `write` command is:

write *filename file_format*

or

write *filename.file_extension*

where *file_extension* indicates the format in which the file is to be saved.

Reading problem files

These commands and options read files into the Interactive Optimizer.

Overview

The `read` command supports a variety of file formats in Interactive Optimizer.

When you are using CPLEX to solve linear optimization problems, you may frequently enter problems by reading them from files instead of entering them from the keyboard. With that practice in view, the following topics continue the tutorial from "Writing problem and solution files" on page 34.

Selecting a read file format

The Interactive Optimizer reads these files formats.

When you type the `read` command in the Interactive Optimizer with the name of a file bearing an extension that it does not recognize, CPLEX displays the following prompt about file formats on the screen:

```
File type options:

alp         LP format problem with generic names and bound annotations
bas         INSERT format basis file
flt         Solution pool filters
```

```
lp          LP format problem file
min         DIMACS min-cost network-flow format file
mps         MPS format problem file
mst         MIP start file
net         CPLEX network-flow format file
ord         Integer priority order file
prm         Non-default parameter file
sav         Binary matrix and basis file
sol         Solution file

File type:
```

Note:

All these file formats are documented in more detail in the *CPLEX File Formats* Reference Manual.

Reading LP files

To read a formatted LP file into the Interactive Optimizer, use this command with options.

At the `CPLEX>` prompt type:

```
read
```

The following message appears requesting a file name:

```
Name of file to read:
```

Four files have been saved at this point in this tutorial:

`example`

`example2`

`example.lp`

`example.bas`

Specify the file named `example` that you saved while practicing the `write` command.

You recall that the example problem was saved in LP format, so in response to the file type prompt, enter:

```
lp
```

CPLEX displays a confirmation message, like this:

```
Problem 'example' read.
Read Time =   0.03 sec.
```

The example problem is now in memory, and you can manipulate it with CPLEX commands.

Tip:

The intermediate prompts for the `read` command can be avoided by entering the entire command on one line, like this:

```
make LPex1.class
java -Djava.library.path=../../../bin/<platform>: \
     -classpath ../../../lib/cplex.jar: LPex1 -r
read example lp
```

Using file extensions

Support for file extensions in the Interactive Optimizer follows these conventions.

If the file name has an extension that corresponds to one of the supported file formats, CPLEX automatically reads it without your having to specify the format. Thus, the following command automatically reads the problem file `example.lp` in LP format:

```
read example.lp
```

Reading MPS files

To read MPS formatted files in the Interactive Optimizer, follow these conventions.

CPLEX can also read industry-standard MPS formatted files. The problem called `afiro.mps` (provided in the CPLEX distribution) serves as an example. If you include the `.mps` extension in the file name, CPLEX recognizes the file as being in MPS format. If you omit the extension, CPLEX attempts to detect whether the file is of a type that it recognizes.

```
read afiro mps
```

After the file has been read, the following message appears:

```
Selected objective sense:  MINIMIZE
Selected objective  name:  obj
Selected RHS        name:  rhs
Problem 'afiro' read.
Read time =    0.01 sec.
```

CPLEX reports additional information when it reads MPS formatted files. Since these files can contain multiple objective function, righthand side, bound, and other information, CPLEX displays which of these is being used for the current problem. See Working with MPS files in the *CPLEX User's Manual* to learn more about special considerations for using MPS formatted files.

Reading basis files

To read basis files into the Interactive Optimizer, follow these conventions.

In addition to other file formats, the `read` command is also used to read basis files. These files contain information for CPLEX that tells the simplex method where to begin the next optimization. Basis files usually correspond to the result of some previous optimization and help to speed re-optimization. They are particularly helpful when you are dealing with very large problems if small changes are made to the problem data.

"Writing basis files" on page 35 showed you how to save a basis file for the `example` after it was optimized. For this tutorial, first read the `example.lp` file. Then read this basis file by typing the following command:

```
read example.bas
```

The message of confirmation:

```
Basis 'example.bas' read.
```

indicates that the basis file was successfully read. If the advanced basis indicator is on, this basis will be used as a starting point for the next optimization, and any new basis created during the session will be used for future optimizations. If the basis changes during a session, you can save it by using the `write` command.

Summary

The general syntax for the read command is:

```
read filename file_format
```

or

```
read filename.file_extension
```

where `file_extension` corresponds to one of the allowed file formats.

Setting CPLEX parameters

The set command controls CPLEX parameters in the Interactive Optimizer.

CPLEX users can vary parameters by means of the `set` command. This command is used to set CPLEX parameters to values different from their default values. The procedure for setting a parameter is similar to that of other commands. Commands can be carried out incrementally or all in one line from the `CPLEX>` prompt. Whenever a parameter is set to a new value, CPLEX inserts a comment in the log file that indicates the new value.

Setting a parameter

To see the parameters that can be changed, type:

```
set
```

The parameters that can be changed are displayed with a prompt, like this:

```
Available Parameters:

advance         set indicator for advanced starting information
barrier         set parameters for barrier optimization
clocktype       set type of clock used to measure time
conflict        set parameters for finding conflicts
defaults        set all parameter values to defaults
dettimelimit    set deterministic time limit in ticks
emphasis        set optimization emphasis
feasopt         set parameters for feasopt
logfile         set file to which results are printed
lpmethod        set method for linear optimization
mip             set parameters for mixed integer optimization
network         set parameters for network optimizations
output          set extent and destinations of outputs
parallel        set parallel optimization mode
preprocessing   set parameters for preprocessing
qpmethod        set method for quadratic optimization
randomseed      set seed to initialize the random number generator
read            set problem read parameters
sifting         set parameters for sifting optimization
simplex         set parameters for primal and dual simplex optimizations
solutiontarget  set type of solution CPLEX will attempt to compute
threads         set default parallel thread count
timelimit       set time limit in seconds
tune            set parameters for parameter tuning
```

```
workdir         set directory for working files
workmem         set memory available for working storage (megabytes)

Parameter to set:
```

If you press the `<return>` key without entering a parameter name, the following message is displayed:

```
No parameters changed.
```

Resetting defaults

After making parameter changes, it is possible to reset all parameters to default values by issuing one command:

```
set defaults
```

This resets all parameters to their default values, except for the name of the log file.

Summary

The general syntax for the `set` command is:

`set` *parameter option new_value*

Displaying parameter settings

The current values of the parameters can be displayed with the command:

```
display settings all
```

A list of parameters with settings that differ from the default values can be displayed with the command:

```
display settings changed
```

For a description of all parameters and their default values, see the reference manual *CPLEX Parameters*.

CPLEX also accepts customized system parameter settings via a parameter specification file. See the *CPLEX File Formats Reference Manual* for a description of the parameter specification file and its use.

Adding constraints and bounds

To add constraints or bounds to a problem in the Interactive Optimizer, use these commands and options.

If you wish to add either new constraints or bounds to your problem, use the `add` command. This command is similar to the `enter` command in the way it is used, but it has one important difference: the `enter` command is used to start a brand new problem, whereas the `add` command only adds new information to the current problem.

Suppose that in the example you need to add a third constraint:

$x1 + 2x2 + 3x3 \geq 50$

You may do either interactively or from a file.

Adding interactively

Type the `add` command, then enter the new constraint on the blank line. After validating the constraint, the cursor moves to the next line. You are in an environment identical to that of the `enter` command after having issued `subject to` . At this point you may continue to add constraints or you may type `bounds` and enter new bounds for the problem. For the present example, type `end` to exit the `add` command. Your session should look like this:

```
add
Enter new constraints and bounds ['end' terminates]:
x1 + 2x2 + 3x3 >= 50
end
Problem addition successful.
```

When the problem is displayed again, the new constraint appears, like this:

```
display problem all

Maximize
 obj: x1 + 2 x2 + 3 x3
Subject To
 c1: - x1 +   x2 +   x3 <= 20
 c2: x1 - 3 x2 +   x3 <= 30
 c3: x1 + 2 x2 + 3 x3 >= 50
Bounds
 0 <= x1 <= 40
 All other variables are >= 0.
end
```

Adding from a file

Alternatively, you may read in new constraints and bounds from a file. If you enter a file name after the `add` command, CPLEX will read a file matching that name. The file contents must comply with standard CPLEX LP format. CPLEX does not prompt for a file name if none is entered. Without a file name, interactive entry is assumed.

Summary

The general syntax for the `add` command is:

add

or

add *filename*

Changing a problem

To change a problem, the Interactive Optimizer supports these commands and options.

Overview

The `change` command modifies a model in the Interactive Optimizer.

The `enter` and `add` commands allow you to build a problem from the keyboard, but they do not allow you to change what you have built. You make changes with the `change` command.

The change command can be used for several different tasks, as demonstrated in the following topics.

What can be changed?

Options of the change command determine what you can modify in a problem entered in the Interactive Optimizer.

Start out by changing the name of the constraint that you added with the add command. In order to see a list of change options, type:

```
change
```

The elements that can be changed are displayed like this:

```
Change options:

bounds      change bounds on a variable
coefficient change a coefficient
delete      delete some part of the problem
name        change a constraint or variable name
objective   change objective function value
problem     change problem type
qpterm      change a quadratic objective term
rhs         change a righthand side or network supply/demand value
sense       change objective function or a constraint sense
type        change variable type
values      change small values in the problem to zero

Change to make:
```

Changing constraint or variable names

These options change the name of a constraint or variable in the Interactive Optimizer.

Enter name at the Change to make: prompt to change the name of a constraint:

```
Change to make: name
```

The present name of the constraint is c3 . In the example, you can change the name to new3 to differentiate it from the other constraints using the following entries:

```
Change a constraint or variable name ['c' or 'v']: c
Present name of constraint: c3
New name of constraint: new3
The constraint 'c3' now has name 'new3'.
```

The name of the constraint has been changed.

The problem can be checked with a display command (for example, display problem constraints new3) to confirm that the change was made.

This same technique can also be used to change the name of a variable.

Changing sense

This option changes the sense of a constraint in the Interactive Optimizer.

Next, change the sense of the new3 constraint from ≥ to ≤ using the sense option of the change command. At the CPLEX> prompt, type:

```
change sense
```

CPLEX prompts you to specify a constraint. There are two ways of specifying this constraint: if you know the name (for example, `new3`), you can enter the name; if you do not know the name, you can specify the index of the constraint. In this example, the index is 3 for the `new3` constraint. Try the first method and type:

```
Change sense of which constraint: new3
Sense of constraint 'new3' is '>='.
```

CPLEX tells you the current sense of the selected constraint. All that is left now is to enter the new sense, which can be entered as `<=` , `>=` , or `=` . You can also type simply `<` (interpreted as ≤) or `>` (interpreted as ≥). The letters `l` , `g` , and `e` are also interpreted as ≤, ≥, and = respectively.

```
New sense ['<=' or '>=' or '=']: <=
Sense of constraint 'new3' changed to '<='.
```

The sense of the constraint has been changed.

The sense of the objective function may be changed by specifying the objective function name (its default is `obj`) or the number 0 when CPLEX prompts you for the constraint. You are then prompted for a new sense. The sense of an objective function can take the value `maximum` or `minimum` or the abbreviation `max` or `min` .

Changing bounds

This option changes the bounds of a variable in the Interactive Optimizer.

When the example was entered, bounds were set specifically only for the variable `x1` . The bounds can be changed on this or other variables with the `bounds` option. Again, start by selecting the command and option.

```
change bounds
```

Select the variable by name or number and then select which bound you would like to change. For the example, change the upper bound of variable `x2` from +∞ to 50.

```
Change bounds on which variable: x2
Present bounds on variable x2: The indicated variable is >= 0.
Change lower or upper bound, or both ['l', 'u', or 'b']: u
Change upper bound to what ['+inf' for no upper bound]: 50
New bounds on variable 'x2':  0 <= x2 <= 50
```

Removing bounds

To remove a bound in the Interactive Optimizer, use the `set` command.

To remove a bound, set it to +∞ or –∞. Interactively, use the identifiers `inf` and `-inf` instead of the symbols. To change the upper bound of `x2` back to +∞, use the one line command:

```
change bounds x2 u inf
```

You receive the message:

```
New bounds on variable 'x2': The indicated variable is >= 0.
```

The bound is now the same as it was when the problem was originally entered.

Changing coefficients of variables

To change the coefficient of a variable in a constraint in the Interactive Optimizer, use options of the `change` command.

Up to this point all of the changes that have been made could be referenced by specifying a single constraint or variable. In changing a coefficient, however, a constraint *and* a variable must be specified in order to identify the correct coefficient. As an example, change the coefficient of x3 in the new3 constraint from 3 to 30.

As usual, you must first specify which change command option to use:

```
change coefficient
```

You must now specify both the constraint row and the variable column identifying the coefficient you wish to change. Enter both the constraint name (or number) and variable name (or number) on the same line, separated by at least one space. The constraint name is new3 and the variable is number 3, so in response to the following prompt, type new3 and 3 , like this, to identify the one to change:

```
Change which coefficient ['constraint' 'variable']: new3 3
Present coefficient of constraint 'new3', variable '3' is 3.000000.
```

The final step is to enter the new value for the coefficient of x3 .

```
Change coefficient of constraint 'new3', variable '3' to what: 30
Coefficient of constraint 'new3', variable '3' changed to 30.000000.
```

Objective and RHS coefficients

The Interactive Optimizer also supports modification of a coefficient in the objective function or righthand side.

To change a coefficient in the objective function, or in the righthand side, use the corresponding option of the change command: objective or rhs. For example, to specify the righthand side of constraint 1 to be 25.0, a user enters the following command (but for this tutorial, do not enter this command now):

```
change rhs 1 25.0
```

Deleting entire constraints or variables

The Interactive Optimizer supports these options of the delete command.

Another option to the change command is delete. This option is used to remove an entire constraint or a variable from a problem. Return the problem to its original form by removing the constraint you added earlier. Type:

```
change delete
```

CPLEX displays a list of delete options.

```
Delete options:

constraints      delete range of constraints
qconstraints     delete range of quadratic constraints
indconstraints   delete range of indicator constraints
soss             delete range of special ordered sets
variables        delete range of variables
filters          delete range of filters
solutions        delete range of solutions from the pool
equality         delete range of equality constraints
greater-than     delete range of greater-than constraints
less-than        delete range of less-than constraints
mipstarts        delete range of mipstarts

Deletion to make:
```

At the first prompt, specify that you want to delete a constraint.

```
 Deletion to make: constraints
```

At the next prompt, enter a constraint name or number, or a range as you did when you used the `display` command. Since the constraint to be deleted is named `new3`, enter that name:

```
Delete which constraint(s): new3
Constraint 3 deleted.
```

Check to be sure that the correct range or number is specified when you perform this operation, since constraints are permanently removed from the problem. Indices of any constraints that appeared after a deleted constraint will be decremented to reflect the removal of that constraint.

The last message indicates that the operation is complete. The problem can now be checked to see if it has been changed back to its original form.

```
display problem all

Maximize
 obj:  x1 + 2 x2 + 3 x3
Subject To
 c1: - x1 +   x2 +   x3 <= 20
 c2:   x1 - 3 x2 +   x3 <= 30
Bounds
 0 <= x1 <= 40
All other variables are >= 0.
```

When you remove a constraint with the `delete` option, that constraint no longer exists in memory; however, variables that appear in the deleted constraint are not removed from memory. If a variable from the deleted constraint appears in the objective function, it may still influence the solution process. If that is not what you want, these variables can be explicitly removed using the `delete` option.

Changing small values to zero

To clean data by zeroing out small values, the Interactive Optimizer offers options for the `change` command.

The `change` command can also be used to clean up data in situations where you know that very small values are not really part of your model but instead are the result of imprecision introduced by finite-precision arithmetic in operations such as round-off.

```
change values
```

CPLEX then prompts you for a tolerance (epsilon value) within which small values should be changed to 0 (zero).

Summary

The general syntax for the `change` command is:

change *option identifier [identifier2] new value*

Executing operating system commands

Operating system commands are accessible from the Interactive Optimizer.

The execute command (`xecute`) is simple but useful. It executes operating system commands outside of the CPLEX environment. By using `xecute`, you avoid having to save a problem and quit CPLEX in order to carry out a system function (such as viewing a directory, for example).

As an example, if you wanted to check whether all of the files saved in the last session are really in the current working directory, the following CPLEX command shows the contents of the current directory in a UNIX operating system, using the UNIX command `ls`:

```
xecute ls -l
total 7448
-r--r--r--    1      3258 Jul 14 10:34 afiro.mps
-rwxr-xr-x    1   3783416 Apr 22 10:32 cplex
-rw-r--r--    1      3225 Jul 14 14:21 cplex.log
-rw-r--r--    1       145 Jul 14 11:32 example
-rw-r--r--    1       112 Jul 14 11:32 example.bas
-rw-r--r--    1       148 Jul 14 11:32 example.lp
-rw-r--r--    1       146 Jul 14 11:32 example2
```

After the command is executed, the `CPLEX>` prompt returns, indicating that you are still in CPLEX. Most commands that can normally be entered from the prompt for your operating system can also be entered with the `xecute` command. The command may be as simple as listing the contents of a directory or printing the contents of a file, or as complex as starting a text editor to modify a file. Anything that can be entered on one line after the operating system prompt can also be executed from within CPLEX. However, this command differs from other CPLEX commands in that it must be entered on a single line. No prompt will be issued. In addition, the operating system may fail to carry out the command. In that case, no message is issued by the operating system, and the result is a return to the `CPLEX>` prompt.

Summary

The general syntax for the `xecute` command is:

`xecute` *`command line`*

Quitting CPLEX

To terminate a session of the Interactive Optimizer, use the `quit` command.

When you are finished using CPLEX and want to leave it, type:

```
quit
```

If a problem has been modified, be sure to save the file before issuing a `quit` command. CPLEX will not prompt you to save your problem.

Advanced features of the Interactive Optimizer

Further reading about advanced features of the Interactive Optimizer is available in the *CPLEX User's Manual*.

This introduction to the Interactive Optimizer presents most of the commands and their options. There are also other, more advanced features of the Interactive Optimizer, documented in the *CPLEX User's Manual*. Here short descriptions of those advanced features and links to further information about them.

The **tuning tool** can help you discern nondefault parameter settings that lead to faster solving time. Examples: time limits on tuning in the Interactive Optimizer shows how to use the tuning tool in the Interactive Optimizer.

The **solution pool** stores multiple solutions to a mixed integer programming (MIP) model. With this feature, you can direct the optimizer to generate multiple solutions in addition to the optimal solution. CPLEX offers facilities to manage the solution pool and to access members of the solution pool. Solution pool: generating and keeping multiple solutions describes those facilities and documents the corresponding commands of the Interactive Optimizer.

The **conflict refiner** diagnoses the cause of infeasibility in a model or MIP start, whether continuous or discrete, whether linear or quadratic. Diagnosing infeasibility by refining conflicts documents the conflict refiner generally, and Meet the conflict refiner in the Interactive Optimizer introduces the conflict refiner as a feature of the Interactive Optimizer.

FeasOpt attempts to repair an infeasibility by modifying the model according to preferences set by the user. FeasOpt accepts an infeasible model and selectively relaxes the bounds and constraints in a way that minimizes a weighted penalty function that you define. Repairing infeasibilities with FeasOpt documents this feature and refers throughout to commands available in the Interactive Optimizer.

The user may supply a **MIP start**, also known as an **advanced start** or a **warm start**, to serve as the first integer solution when CPLEX solves a MIP. Such a solution might come from a MIP problem solved previously or from the user's knowledge of the problem, for example. MIP starts and the Interactive Optimizer introduces commands of the Interactive Optimizer to manage MIP starts.

Chapter 4. Concert Technology tutorial for C++ users

This tutorial shows you how to write C++ applications using CPLEX with Concert Technology. In this tutorial you will learn about:

The design of CPLEX in Concert Technology C++ applications

CPLEX objects are necessary to an application in C++.

A clear understanding of C++ **objects** is fundamental to using Concert Technology with CPLEX to build and solve optimization models. These objects can be divided into two categories:

1. **Modeling objects** are used to define the optimization problem. Generally an application creates multiple modeling objects to specify one optimization problem. Those objects are grouped into an IloModel **object** representing the complete optimization problem.
2. `IloCplex` **objects** are used to solve the problems that have been 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.

Thus, the modeling and optimization parts of a user-written application program are represented by a group of interacting C++ objects created and controlled within the application. Figure 3 shows a picture of an application using CPLEX with Concert Technology to solve optimization problems.

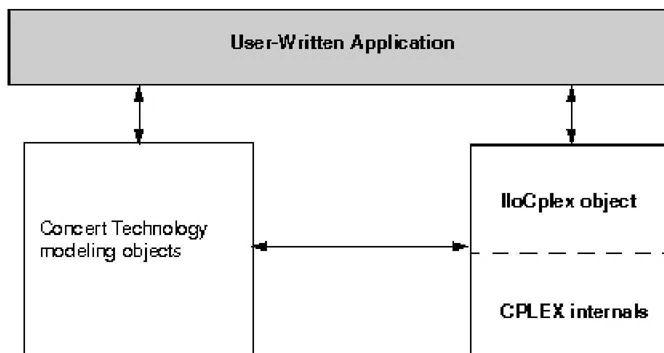

Figure 3. A View of CPLEX with Concert Technology

The CPLEX internals include the computing environment, its communication channels, and your problem objects.

This brief tutorial introduces the modeling and solution classes provided by Concert Technology and CPLEX. More information about the algorithm class `IloCplex` and its nested classes can be found in the *CPLEX User's Manual* and *CPLEX C++ API Reference Manual*.

© Copyright IBM Corp. 1987, 2013

Compiling CPLEX in Concert Technology C++ applications

When you compile a C++ application with a C++ library like CPLEX in Concert Technology, you need to tell your *compiler* where to find the CPLEX and Concert include files (that is, the header files), and you also need to tell the *linker* where to find the CPLEX and Concert libraries. The sample projects and `makefiles` illustrate how to carry out these crucial steps for the examples in the standard distribution. They use relative path names to indicate to the compiler where the header files are, and to the linker where the libraries are.

Testing your installation on UNIX

After you install CPLEX on a UNIX platform, you can test the installation.

About this task

To run the test, follow these steps.

Procedure

1. First check the file `readme.html` in the standard distribution to locate the right subdirectory containing a `makefile` appropriate for your platform.
2. Go to that subdirectory.
3. Then use the sample `makefile` located there to compile and link the examples that came in the standard distribution.

   `make all` compiles and links examples for all of the APIs.

   `make all_cpp` compiles and links the examples of the C++ API.
4. Execute one of the compiled examples.

   `make execute_all` executes all of the examples.

   `make execute_cpp` executes only the C++ examples.

Testing your installation on Windows

After you install CPLEX on a Windows platform, you can test the installation.

To run the test on a Windows platform, first consult the file `c_cpp.html` in the standard distribution. Then follow the directions you find there.

The examples have been tested repeatedly on all the platforms compatible with CPLEX, so if you successfully compile, link, and execute them, then you can be sure that your installation is correct.

In case of problems

CPLEX offers trouble-shooting procedures.

If you encounter difficulty when you try the installation test, then there is a problem in your installation, and you need to correct it before you begin real work with CPLEX.

For example, if you get a message from the compiler such as

```
ilolpex3.cpp 1: Can't find include file ilcplex/ilocplex.h
```

then you need to verify that your compiler knows where you have installed CPLEX and its include files (that is, its header files).

If you get a message from the linker, such as

```
ld: -lcplex: No such file or directory
```

then you need to verify that your linker knows where the CPLEX library is located on your system.

If you successfully compile, link, and execute one of the examples in the standard distribution, then you can be sure that your installation is correct, and you can begin to use CPLEX with Concert Technology seriously.

The anatomy of a Concert Technology C++ application

Concert Technology is a C++ class library, and therefore Concert Technology applications consist of interacting C++ objects. This topic gives a short introduction to the most important classes that are usually found in a complete Concert Technology CPLEX C++ application.

Constructing the environment: IloEnv

The class IloEnv constructs a CPLEX environment.

An environment, that is, an instance of `IloEnv` is typically the first object created in any Concert Technology application.

You construct an `IloEnv` object by declaring a variable of type `IloEnv` . For example, to create an environment named `env` , you do this:

```
IloEnv env;
```

Note:

The environment object created in a Concert Technology application is different from the environment created in the CPLEX C library by calling the routine `CPXopenCPLEX`.

The environment object is of central importance and needs to be available to the constructor of all other Concert Technology classes because (among other things) it provides optimized memory management for objects of Concert Technology classes. This provides a boost in performance compared to the memory management of the operating system.

As is the case for most Concert Technology classes, `IloEnv` is a *handle class*. This means that the variable `env` is a pointer to an implementation object, which is created at the same time as `env` in the above declaration. One advantage of using handles is that if you assign handle objects, all that is assigned is a pointer. So the statement

```
IloEnv env2 = env;
```

creates a second handle pointing to the implementation object that `env` already points to. Hence there may be an arbitrary number of `IloEnv` handle objects all pointing to the same implementation object. When terminating the Concert Technology application, the implementation object must be destroyed as well. This must be done explicitly by the user by calling

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

for just *ONE* of the `IloEnv` handles pointing to the implementation object to be destroyed. The call to `env.end` is generally the last Concert Technology operation in an application.

Creating a model: IloModel

CPLEX C++ offers modeling objects for an application.

After creating the environment, a Concert application is ready to create one or more optimization models. Doing so consists of creating a set of modeling objects to define each optimization model.

Modeling objects, like IloEnv objects, are handles to implementation objects. Though you will be dealing only with the handle objects, it is the implementation objects that contain the data that specifies the optimization model. If you need to remove an implementation object from memory, you need to call the `end` method for one of its handle objects.

Modeling objects are also known as *extractables* because it is the individual modeling objects that are extracted one by one when you extract an optimization model to IloCplex . So, extractables are characterized by the possibility of being extracted to algorithms such as `IloCplex` . In fact, they all are inherited from the class `IloExtractable` . In other words, `IloExtractable` is the base class of all classes of extractables or modeling objects.

The most fundamental extractable class is `IloModel` . Objects of this class are used to define a complete optimization model that can later be extracted to an IloCplex object. You create a model by constructing an object of type IloModel. For example, to construct a modeling object named `model`, within an existing environment named `env`, you write the following line:

```
IloModel model(env);
```

At this point, it is important to note that the environment is passed as an argument to the constructor. There is also a constructor that does not use the environment argument, but this constructor creates an empty handle, the handle corresponding to a `NULL` pointer. Empty handles cannot be used for anything but for assigning other handles to them. Unfortunately, it is a common mistake to try to use empty handles for other things.

After an IloModel object has been constructed, it is populated with the extractables that define the optimization model. The most important classes here are:

IloNumVar	representing modeling variables;
IloRange	defining constraints of the form l <= expr <= u, where expr is a linear expression; and
IloObjective	representing an objective function.

You create objects of these classes for each variable, constraint, and objective function of your optimization problem. Then you add the objects to the model by calling

```
model.add(object);
```

for each extractable `object` . There is no need to explicitly add the variable objects to a model, as they are implicitly added when they are used in the range constraints (instances of `IloRange`) or the objective. At most one objective can be used in a model with IloCplex .

Modeling variables are constructed as objects of class IloNumVar , by defining variables of type IloNumVar . Concert Technology provides several constructors for doing this; the most flexible form is:

```
IloNumVar x1(env, 0.0, 40.0, ILOFLOAT);
```

This definition creates the modeling variable x1 with lower bound 0.0, upper bound 40.0 and type ILOFLOAT , which indicates the variable is continuous. Other possible variable types include ILOINT for integer variables and ILOBOOL for Boolean variables.

For each variable in the optimization model a corresponding object of class IloNumVar must be created. Concert Technology provides a wealth of ways to help you construct all the IloNumVar objects.

After all the modeling variables have been constructed, they can be used to build expressions, which in turn are used to define objects of class IloObjective and IloRange . For example,

```
IloObjective obj = IloMinimize(env, x1 + 2*x2 + 3*x3);
```

This creates the extractable obj of type IloObjective which represents the objective function of the example presented in "Introducing CPLEX" on page v.

Consider in more detail what this line does. The function IloMinimize takes the environment and an expression as arguments, and constructs a new IloObjective object from it that defines the objective function to minimize the expression. This new object is returned and assigned to the new handle obj .

After an objective extractable is created, it must be added to the model. As noted above this is done with the add method of IloModel . If this is all that the variable obj is needed for, it can be written more compactly, like this:

```
model.add(IloMinimize(env, x1 + 2*x2 + 3*x3));
```

This way there is no need for the program variable obj and the program is shorter. If in contrast, the objective function is needed later, for example, to change it and re-optimize the model when doing scenario analysis, the variable obj must be created in order to refer to the objective function. (From the standpoint of algorithmic efficiency, the two approaches are comparable.)

Creating constraints and adding them to the model can be done just as easily with the following statement:

```
model.add(-x1 + x2 + x3 <= 20);
```

The part -x1 + x2 + x3 <= 20 creates an object of class IloRange that is immediately added to the model by passing it to the method IloModel::add . Again, if a reference to the IloRange object is needed later, an IloRange handle object must be stored for it. Concert Technology provides flexible array classes for storing data, such as these IloRange objects. As with variables, Concert Technology provides a variety of constructors that help create range constraints.

While those examples use expressions with modeling variables directly for modeling, it should be pointed out that such expressions are themselves represented by yet another Concert Technology class, IloExpr . Like most Concert Technology objects, IloExpr objects are handles. Consequently, the method end must be called when the object is no longer needed. The only exceptions are implicit expressions, where the user does not create an IloExpr object, such as

when writing (for example) `x1 + 2*x2` . For such implicit expressions, the method `end` should not be called. The importance of the class `IloExpr` becomes clear when expressions can no longer be fully spelled out in the source code but need instead to be built up in a loop. Operators like `+=` provide an efficient way to do this.

Solving the model: IloCplex

The class IloCplex solves a model.

After the optimization problem has been created in an `IloModel` object, it is time to create the IloCplex object for solving the problem by creating an instance of the class `IloCplex`. For example, to create an object named `cplex`, write the following line:

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

again using the environment `env` as an argument. The CPLEX object can then be used to extract the model to be solved. One way to extract the model is to call `cplex.extract(model)`. However, experienced Concert users practice a shortcut that performs the construction of the `cplex` object and the extraction of the model in one line:

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

This shortcut works because the modeling object `model` contains within it the reference to the environment named `env`.

After this line, the object `cplex` is ready to solve the optimization problem defined by `model`. To solve the model, call:

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

This method returns an `IloBool` value, where `IloTrue` indicates that `cplex` successfully found a feasible (yet not necessarily optimal) solution, and `IloFalse` specifies that no solution was found. More precise information about the outcome of the last call to the method IloCplex::solve can be obtained by calling:

```
cplex.getStatus ();
```

The returned value tells you what CPLEX found out about the model: whether it found the optimal solution or only a feasible solution, whether it proved the model to be unbounded or infeasible, or whether nothing at all has been proved at this point. Even more detailed information about the termination of the solve call is available through method IloCplex::getCplexStatus.

Querying results

Query methods access information about the solution.

After successfully solving the optimization problem, you probably are interested in accessing the solution. The following methods can be used to query the solution value for a variable or a set of variables:

```
IloNum IloCplex::getValue (IloNumVar var) const;
void IloCplex::getValues (IloNumArray val,
                          const IloNumVarArray var) const;
```

For example:

```
IloNum val1 = cplex.getValue(x1);
```

stores the solution value for the modeling variable x1 in val1. Other methods are available for querying other solution information. For example, the objective function value of the solution can be accessed using:

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

Handling errors

A robust application of CPLEX in C++ avoids errors with assertions and handles unforeseeable errors with exceptions.

Concert Technology provides two lines of defense for dealing with error conditions, suited for addressing two kinds of errors. The first kind covers simple programming errors. Examples of this kind are trying to use empty handle objects or passing arrays of incompatible lengths to functions.

This kind of error is usually an oversight and should not occur in a correct program. In order not to pay any runtime cost for correct programs asserting such conditions, the conditions are checked using `assert` statements. The checking is disabled for production runs if compiled with the `-DNDEBUG` compiler option.

The second kind of error is more complex and cannot generally be avoided by correct programming. An example is memory exhaustion. The data may simply require too much memory, even when the program is correct. This kind of error is always checked at runtime. In cases where such an error occurs, Concert Technology throws a C++ exception.

In fact, Concert Technology provides a hierarchy of exception classes that all derive from the common base class `IloException` . Exceptions derived from this class are the only kind of exceptions that are thrown by Concert Technology. The exceptions thrown by `IloCplex` objects all derive from class `IloAlgorithm::Exception` or IloCplex::Exception .

To handle exceptions gracefully in a Concert Technology application, include all of the code in a `try/catch` clause, like this:

```
IloEnv env;
try {
// ...
} catch (IloException& e) {
cerr << "Concert Exception: " << e << endl;
} catch (...) {
cerr << "Other Exception" << endl;
}
env.end();
```

Note:

The construction of the environment comes before the `try/catch` clause. In case of an exception, `env.end` must still be called. To protect against failure during the construction of the environment, you can add another `try/catch` clause.

If code other than Concert Technology code is used in the part of that sample denoted by `...` , all other exceptions will be caught with the statement `catch(...)` . Doing so is good practice, as it makes sure that no exception is unhandled.

Building and solving a small LP model in C++

This sample solves a linear programming model in C++.

Overview

The sample illustrates three alternative approaches: modeling by rows, columns, or nonzero elements.

This sample offers a complete example of building and solving a small LP model. It demonstrates:

- “Modeling by rows” on page 57
- “Modeling by columns” on page 57
- “Modeling by nonzero elements” on page 58

This example `ilolpex1.cpp` is one of the sample applications in the standard CPLEX distribution. It is an extension of the example presented in “Introducing CPLEX” on page v. It shows three different ways of creating a Concert Technology LP model, as well as how to solve it using IloCplex, and how to access the solution. Here is the problem that the example optimizes:

Maximize	$x_1 + 2x_2 + 3x_3$
subject to	
	$-x_1 + x_2 + x_3 \le 20$
	$x_1 - 3x_2 + x_3 \le 30$
with these bounds	
	$0 \le x_1 \le 40$
	$0 \le x_2 \le infinity$
	$0 \le x_3 \le infinity$

The first operation is to create the environment object `env` , and the last operation is to destroy it by calling `env.end` . The rest of the code is enclosed in a `try/catch` clause to gracefully handle any errors that may occur.

First the example creates the model object and, after checking the correctness of command line arguments, it creates empty arrays for storing the variables and range constraints of the optimization model. Then, depending on the command line argument, the example calls one of the functions `populatebyrow`, `populatebycolumn`, or `populatebynonzero`, to fill the model object with a representation of the optimization problem. These functions place the variable and range objects in the arrays `var` and `con` which are passed to them as arguments.

After the model has been populated, the IloCplex algorithm object `cplex` is created and the model is extracted to it. The following call of the method `solve` invokes the optimizer. If it fails to generate a solution, an error message is issued to the error stream of the environment, `cplex.error()` , and the integer -1 is thrown as an exception.

IloCplex provides the output streams `out` for general logging, `warning` for warning messages, and `error` for error messages. They are preconfigured to `cout` , `cerr` , and `cerr` respectively. Thus by default you will see logging output on the screen when invoking the method solve. This can be turned off by calling `cplex.setOut(env.getNullStream())` , that is, by redirecting the `out` stream of the IloCplex object `cplex` to the null stream of the environment.

If a solution is found, solution information is output through the channel, `env.out` which is initialized to `cout` by default. The output operator << is defined for type `IloAlgorithm::Status` as returned by the call to getStatus . It is also defined for

IloNumArray , the Concert Technology class for an array of numerical values, as returned by the calls to getValues , getDuals , getSlacks , and getReducedCosts. In general, the output operator is defined for any Concert Technology array of elements if the output operator is defined for the elements.

The functions named populateby * are purely about modeling and are completely decoupled from the algorithm IloCplex. In fact, they don't use the cplex object, which is created only after executing one of these functions.

Modeling by rows

CPLEX supports the approach of modeling by rows in the sample.

The function populatebyrow creates the variables and adds them to the array x. Then the objective function and the constraints are created using expressions over the variables stored in x. The range constraints are also added to the array of constraints c. The objective and the constraints are added to the model.

Modeling by columns

CPLEX also supports the approach of modeling by columns in the sample.

The function populatebycolumn can be viewed as the transpose of populatebyrow. While, for simple examples like this one, population by rows may seem the most straightforward and natural approach, there are some models where modeling by column is a more natural or more efficient approach.

When an application creates a model by columns, range objects are created with their lower and upper bound only. No expression is given since the variables are not yet created. Similarly, the objective function is created with only its intended optimization sense, and without any expression. Next the variables are created and installed in the already existing ranges and objective.

Column expressions provide the description of how the newly created variables are to be installed in the ranges and objective. These column expressions are represented by the class IloNumColumn. Column expressions consist of objects of the class IloAddNumVar linked together with the operator +. These IloAddNumVar objects are created using the operator() of the classes IloObjective and IloRange. These overloaded operators define how to install a new variable to the invoking objective or range objects. For example, obj(1.0) creates an instance of IloAddNumVar capable of adding a new modeling variable with a linear coefficient of 1.0 to the expression in obj. Column expressions can be built in loops by means of the operator +=.

Column expressions (objects of the class IloNumColumn) are handle objects, like most other Concert Technology objects. The method end must therefore be called to delete the associated implementation object when it is no longer needed. However, for implicit column expressions, where no IloNumColumn object is explicitly created, such as the ones used in this example, the method end should not be called.

The column expression is passed as an argument to the constructor of the class IloNumVar. For example the constructor IloNumVar(obj(1.0) + c[0](-1.0) + c[1](1.0), 0.0, 40.0) creates a new modeling variable with lower bound 0.0, upper bound 40.0 and, by default, type ILOFLOAT, and adds it to the objective obj with a linear coefficient of 1.0, to the range c[0] with a linear coefficient of -1.0 and to

c[1] with a linear coefficient of 1.0. Column expressions can be used directly to construct numeric variables with default bounds [0, IloInfinity] and type ILOFLOAT, as in the following statement:

```
x.add(obj(2.0) + c[0]( 1.0) + c[1](-3.0));
```

where IloNumVar does not need to be explicitly written. Here, the C++ compiler recognizes that an IloNumVar object needs to be passed to the add method and therefore automatically calls the constructor IloNumVar(IloNumColumn) in order to create the variable from the column expression.

Modeling by nonzero elements

CPLEX supports the approach of modeling by nonzero elements in the sample.

The last of the three functions that can be used to build the model is populatebynonzero. It creates objects for the objective and the ranges without expressions, and variables without columns. The methods IloObjective::setLinearCoef, setLinearCoefs, and IloRange::setLinearCoef, setLinearCoefs are used to set individual nonzero values in the expression of the objective and the range constraints. As usual, the objective and ranges must be added to the model.

You can view the complete program online in the standard distribution of the product at *yourCPLEXinstallation* /examples/src/ilolpex1.cpp.

Writing and reading models and files

CPLEX supports reading models from files and writing models to files in a C++ application.

In example ilolpex1.cpp , one line is still unexplained:

```
cplex.exportModel ("lpex1.lp");
```

This statement causes cplex to write the model it has currently extracted to the file called lpex1.lp . In this case, the file will be written in LP format. (That format is documented in the reference manual *CPLEX File Formats*.) Other formats supported for writing problems to a file are MPS and SAV (also documented in the reference manual *CPLEX File Formats*). IloCplex decides which file format to write based on the extension of the file name.

IloCplex also supports reading of files through one of its importModel methods. A call to importModel causes CPLEX to read a problem from the file file.lp and add all the data in it to model as new objects. (Again, MPS and SAV format files are also supported.) In particular, CPLEX creates an instance of

IloObjective	for the objective function found in file.lp ,
IloNumVar	for each variable found in file.lp , except
IloSemiContVar	for each semi-continuous or semi-integer variable found in file.lp,
IloRange	for each row found in file.lp ,
IloSOS1	for each SOS of type 1 found in file.lp , and
IloSOS2	for each SOS of type 2 found in file.lp .

If you also need access to the modeling objects created by importModel , two additional signatures are provided:

```
void IloCplex::importModel (IloModel& m,
                            const char* filename,
                            IloObjective& obj,
                            IloNumVarArray vars,
                            IloRangeArray  rngs) const;
```

and

```
void IloCplex::importModel (IloModel& m,
                            const char* filename,
                            IloObjective& obj,
                            IloNumVarArray vars,
                            IloRangeArray rngs,
                            IloSOS1Array sos1,
                            IloSOS2Array sos2) const;
```

They provide additional arguments so that the newly created modeling objects will be returned to the caller. Example program `ilolpex2.cpp` shows how to use the method `importModel`.

Selecting an optimizer

Select an optimizer in a C++ application on the basis of the problem type.

IloCplex treats all problems it solves as Mixed Integer Programming (MIP) problems. The algorithm used by `IloCplex` for solving MIP is known as dynamic search or branch and cut (referred to in some contexts as branch and bound) and is documented in more detail in the *CPLEX User's Manual*. For this tutorial, it is sufficient to know that this algorithm consists of solving a sequence of LPs, QPs, or QCPs that are generated in the course of the algorithm. The first LP, QP, or QCP to be solved is known as the root, while all the others are referred to as nodes and are derived from the root or from other nodes. If the model extracted to the `cplex` object is a pure LP, QP, or QCP (no integer variables), then it will be fully solved at the root.

As mentioned in "Optimizer options" on page vii, various optimizer options are provided for solving LPs, QPs, and QCPs. While the default optimizer works well for a wide variety of models, `IloCplex` allows you to control which option to use for solving the root and for solving the nodes, respectively, by the following methods:

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

where IloCplex::Algorithm is an enumeration type. It defines the following symbols with their meaning:

`IloCplex::AutoAlg`	allow CPLEX to choose the algorithm
`IloCplex::Dual`	use the dual simplex algorithm
`IloCplex::Primal`	use the primal simplex algorithm
`IloCplex::Barrier`	use the barrier algorithm
`IloCplex::Network`	use the network simplex algorithm for the embedded network
`IloCplex::Sifting`	use the sifting algorithm
`IloCplex::Concurrent`	allow CPLEX to use multiple algorithms on multiple computer processors

For QP models, only the `AutoAlg`, `Dual`, `Primal`, `Barrier`, and `Network` algorithms are applicable.

Set the root algorithm argument to select the optimizer that CPLEX uses to solve a pure LP or QPs. The example `ilolpex2.cpp` illustrates this practice.

Reading a problem from a file: example ilolpex2.cpp

The sample `ilolpex2.cpp` illustrates reading a model from a file in a C++ application.

Overview

The sample `ilolpex2.cpp` reads a model from a file, solves the problem with a specified optimizer, prints the basis, and accesses quality information about the solution.

This example shows how to read an optimization problem from a file and solve it with a specified optimizer. It prints solution information, including a simplex basis, if available. Finally, the application prints the maximum infeasibility of any variable of the solution.

The file to read and the optimizer choice are passed to the program via command line arguments. For example, this command:

```
ilolpex2 example.mps d
```

reads the file `example.mps` and solves the problem with the dual simplex optimizer.

Example `ilolpex2` demonstrates:

- “Reading the model from a file”
- “Selecting the optimizer”
- “Accessing basis information” on page 61
- “Querying quality measures” on page 61

The general structure of this example is the same as for example `ilolpex1.cpp` . It starts by creating the environment and terminates with destroying it by calling the `end` method. The code in between is enclosed in `try/catch` statements for error handling.

You can view the complete program online in the standard distribution of the product at *yourCPLEXinstallation* `/examples/src/ilolpex2.cpp` .

Reading the model from a file

The sample reads a model from a file for extraction later.

The model is created by reading it from the file specified as the first command line argument `argv[1]` . The method importModel of an `IloCplex` object reads the model from the file but it does not extract the model for solution. That is, in this case, the IloCplex object is used as a model reader rather than an optimizer. Calling `importModel` does not extract the model to the invoking `cplex` object. Consequently, extraction must be done later by a call to `cplex.extract(model)`. The objects `obj` , `var` , and `rng` are passed to importModel so that later on, when results are queried, the variables will be accessible.

Selecting the optimizer

The sample provides a command line argument for the application user to select an optimizer.

The switch statement controlled by the second command line argument selects the optimizer. A user of the application specifies the optimizer by means of a CPLEX parameter. In the sample, a call to `setParam(IloCplex::RootAlg, alg)` specifies the selected optimizer as an instance of `IloCplex::Algorithm`.

Accessing basis information

The sample may query basis information from the solution, if basis information is available.

After the sample solves the model by calling the method solve, it accesses the results in the same way as in `ilolpex1.cpp`, with the exception of basis information for the variables. Not all optimizers compute basis information. In other words, not all solutions can be queried for basis information. In particular, basis information is not available when the model is solved by the barrier optimizer (`IloCplex::Barrier`) without crossover (parameter `IloCplex::BarCrossAlg` set to `IloCplex::NoAlg`).

Querying quality measures

The sample shows how to access values measuring the quality of a solution.

Finally, the program prints the maximum primal infeasibility or bound violation of the solution. To cope with the finite precision of the numerical computations done on the computer, IloCplex allows some tolerances by which (for instance) optimality conditions may be violated. A long list of other quality measures is available.

Modifying and re-optimizing

CPLEX offers means to modify the model and re-optimize, using available information from previous optimizations.

In many situations, the solution to a model is only the first step. One of the important features of Concert Technology is the ability to modify and then re-solve the model even after it has been extracted and solved one or more times.

A look back to examples `ilolpex1.cpp` and `ilolpex2.cpp` reveals that models have been modified all along. Each time an extractable object is added to a model, it changes the model. However, those examples made all such changes before the model was extracted to CPLEX.

Concert Technology maintains a link between the model and all IloCplex objects that may have extracted it. This link is known as *notification*. Each time a modification of the model or one of its extractable objects occurs, the `IloCplex` objects that extracted the model are notified about the change. They then track the modification in their internal representations.

Moreover, IloCplex tries to maintain as much information from a previous solution as is possible and reasonable, when the model is modified, in order to have a better start when solving the modified model. In particular, when solving LPs or QPs with a simplex method, `IloCplex` attempts to maintain a basis which will be used the next time the method solve is invoked, with the aim of making subsequent solves go faster.

Modifying an optimization problem: example ilolpex3.cpp

The sample `ilolpex3.cpp` shows how to modify a model so that CPLEX can re-optimize and re-use any information available from previous optimizations.

Overview

The example `ilolpex3.cpp` is based on a network flow model.

This example demonstrates:

- "Setting CPLEX parameters" on page 63
- "Modifying an optimization problem" on page 63
- "Starting from a previous basis" on page 63

Here is the problem that example `ilolpex3` solves:

Minimize c^Tx

subject to

$Hx = d$

$Ax = b$

$l \le x \le u$

where

$$H = \begin{pmatrix} -1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & -1 \end{pmatrix} \quad d = \begin{pmatrix} -3 \\ 1 \\ 4 \\ 3 \\ -5 \end{pmatrix}$$

$$A = \begin{pmatrix} 2 & 1 & -2 & -1 & 2 & -1 & -2 & -3 \\ 1 & -3 & 2 & 3 & -1 & 2 & 1 & 1 \end{pmatrix} \quad b = \begin{pmatrix} 4 \\ -2 \end{pmatrix}$$

$$c = (-9\ 1\ 4\ 2\ -8\ 2\ 8\ 12)$$

$$l = (0\ 0\ 0\ 0\ 0\ 0\ 0\ 0)$$

$$u = (50\ 50\ 50\ 50\ 50\ 50\ 50\ 50)$$

The constraints *Hx=d* represent the flow conservation of a pure network flow. The example solves this problem in two steps:

1. The CPLEX network optimizer is used to solve

Minimize c^Tx

subject to

$Hx = d$

$l \le x \le u$

2. The constraints *Ax=b* are added to the problem, and the dual simplex optimizer is used to solve the full problem, starting from the optimal basis of the network problem. The dual simplex method is highly effective in such a case because this basis remains dual feasible after the slacks (artificial variables) of the added constraints are initialized as basic.

Notice that the `0` (zero) values in the data are omitted in the example program. CPLEX makes extensive use of sparse matrix methods and, although CPLEX correctly handles any explicit zero coefficients given to it, most programs solving models of more than modest size benefit (in terms of both storage space and speed) if the natural sparsity of the model is exploited from the very start.

Before the model is solved, the network optimizer is selected by setting the `RootAlg` parameter to the value `IloCplex::Network,` as shown in example `ilolpex2.cpp` . The simplex display parameter SimDisplay (documented in the reference manual of CPLEX parameters as simplex iteration information display) is set so that the simplex algorithm issues logging information as it executes.

Setting CPLEX parameters

The sample shows how to set CPLEX parameters in a C++ application..

IloCplex provides a variety of parameters that allow you to control the solution process. They can be categorized as Boolean, integer, numeric, and string parameters. They are represented by the enumeration types IloCplex::BoolParam, IloCplex::IntParam, IloCplex::NumParam, and IloCplex::StringParam, respectively.

Modifying an optimization problem

The sample emphasizes modification with respect to extraction and notification.

After the simple model is solved and the resulting objective value is passed to the output channel `cplex.out`, the remaining constraints are created and added to the model. At this time the model has already been extracted to the object `cplex`. As a consequence, whenever the model is modified by adding a constraint, this addition is immediately reflected in the object `cplex` via notification.

Starting from a previous basis

The sample illustrates using an optimal basis from a previous solution.

Before solving the modified problem, example `ilolpex3.cpp` sets the optimizer option to Dual, as this is the algorithm that can generally take best advantage of the optimal basis from the previous solve after the addition of constraints.

Complete program

The sample is available online.

You can view the complete program online in the standard distribution of the product at *yourCPLEXinstallation* `/examples/src/ilolpex3.cpp` .

Chapter 5. Concert Technology tutorial for Java users

Applications written in the Java programming language use CPLEX with Concert Technology in the Java API.

Overview

A typical application with CPLEX in Java includes these features.

Concert Technology allows your application to call IBM ILOG CPLEX directly, through the Java Native Interface (JNI). This Java interface supplies a rich means for you to use Java objects to build your optimization model.

The class IloCplex implements the Concert Technology interface for creating variables and constraints. It also provides functionality for solving Mathematical Programing (MP) problems and accessing solution information.

Compiling CPLEX in Concert Technology Java applications

When you compile CPLEX in a Java application, you need to specify the location of the CPLEX JAR in the classpath.

Paths and JARs

Your Java classpath is important to CPLEX in Java applications.

When compiling a Java application that uses Concert Technology, you need to inform the Java compiler where to find the file `cplex.jar` containing the CPLEX Concert Technology class library. To do this, you add the `cplex.jar` file to your classpath. This is most easily done by passing the command-line option to the Java compiler `javac`, like this:

`–classpath` *path_to_*`cplex.jar`

If you need to include other Java class libraries, you should add the corresponding `jar` files to the classpath as well. Ordinarily, you should also include the current directory (`.`) to be part of the Java classpath.

At execution time, the same classpath setting is needed. Additionally, since CPLEX is implemented via JNI, you need to instruct the Java Virtual Machine (JVM) where to find the shared library (or dynamic link library) containing the native code to be called from Java. You indicate this location with the command line option:

`-Djava.library.path=`*path_to_shared_library*

to the `java` command. Note that, unlike the `cplex.jar` file, the shared library is system-dependent; thus the exact path name of the location for the library to be used may differ depending on the platform you are using.

Adapting build procedures to your platform

CPLEX makefiles and other aids support Java application development with CPLEX.

© Copyright IBM Corp. 1987, 2013

About this task

Pre-configured compilation and runtime commands are provided in the standard distribution, through the UNIX makefiles and Windows `javamake` file for `Nmake` . However, these scripts presume a certain relative location for the files already mentioned; for application development, most users will have their source files in some other location.

Here are suggestions for establishing build procedures for your application.

Procedure

1. First check the `readme.html` file in the standard distribution, under the Supported Platforms heading to locate the *machine* and *libformat* entry for your UNIX platform, or the compiler and library-format combination for Windows.
2. Go to the subdirectory in the `examples` directory where CPLEX is installed on your machine. On UNIX, this will be *machine/libformat*, and on Windows it will be *compiler\libformat*. This subdirectory will contain a `makefile` or `javamake` appropriate for your platform.
3. Then use this file to compile the examples that came in the standard distribution by calling `make execute_java` (UNIX) or `nmake -f javamake execute` (Windows).
4. Carefully note the locations of the needed files, both during compilation and at run time, and convert the relative path names to absolute path names for use in your own working environment.

In case problems arise

CPLEX supports trouble-shooting procedures specific to Java applications.

About this task

If a problem occurs in the compilation phase, make sure your Java compiler is correctly set up and that your `classpath` includes the `cplex.jar` file.

If compilation is successful and the problem occurs when executing your application, there are three likely causes:

Procedure

1. If you get a message like `java.lang.NoClassDefFoundError` your `classpath` is not correctly set up. Make sure you use `-classpath <path_to_cplex.jar>` in your `java` command.
2. If you get a message like `java.lang.UnsatisfiedLinkError`, you need to set up the path correctly so that the JVM can locate the CPLEX shared library. Make sure you use the following option in your `java` command:

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

The design of CPLEX in Concert Technology Java applications

Your Java application includes CPLEX as a component.

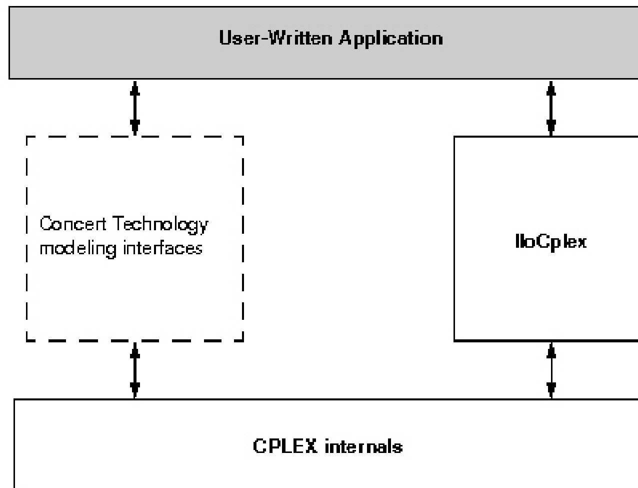

Figure 4. A View of CPLEX in Concert Technology

Figure 4 illustrates the design of Concert Technology and how a user-application uses it. Concert Technology defines a set of interfaces for modeling objects. Such interfaces do not actually consume memory. (For this reason, the box in the figure has a dotted outline.) When a user creates a Concert Technology modeling object using CPLEX, an object is created in CPLEX to implement the interface defined by Concert Technology. However, a user application never accesses such objects directly but only communicates with them through the interfaces defined by Concert Technology.

For more detail about these ideas, see the *CPLEX User's Manual*, especially the topic Concert Technology for Java users.

The anatomy of a Concert Technology Java application

A Java application of CPLEX includes these parts.

Structure of an application

Java applications using CPLEX observe object-oriented conventions.

To use the CPLEX Java interfaces, you need to import the appropriate packages into your application with these lines:

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

As for every Java application, a CPLEX application is implemented as a method of a class. In this discussion, the method will be the static `main` method. The first task is to create an IloCplex object. It is used to create all the modeling objects needed to represent the model. For example, an integer variable with bounds 0 and 10 is created by calling `cplex.intVar(0, 10)` , where `cplex` is the IloCplex object.

Since Java error handling in CPLEX uses exceptions, you should include the Concert Technology part of an application in a `try /catch` statement. All the exceptions thrown by any Concert Technology method are derived from IloException. Thus IloException should be caught in the `catch` statement.

In summary, here is the structure of a Java application that calls CPLEX:

```
-classpath <path_to_cplex.jar>

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

Create the model

Java methods create a model in a Java application of CPLEX.

The IloCplex object provides the functionality to create an optimization model that can be solved with IloCplex. The class `IloCplex` implements the Concert Technology interface IloModeler and its extensions IloMPModeler and IloCplexModeler. These interfaces define the constructors for modeling objects of the following types, which can be used with `IloCplex` :

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

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

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

There is a wealth of other methods for creating arrays or individual modeling variables. The documentation for IloModeler, IloCplexModeler, and IloMPModeler gives you the complete list.

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

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

can be created like this:

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

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

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

The advantage of using IloLinearNumExpr over the first way is that you can more easily build up your linear expression in a loop, which is what is typically needed in more complex applications. The interface IloLinearNumExpr is an extension of IloNumExpr and thus can be used anywhere an expression can be used.

As mentioned before, expressions can be used to create constraints or an objective function for a model. Here is how to create a minimization objective for that expression:

In addition to your creating an objective, you must also instruct IloCplex to use that objective in the model it solves. To do so, *add* the objective to `IloCplex` like this:

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

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

There is a shortcut notation for creating and adding the objective: `cplex.addMinimize(expr);`This shortcut uses the method addMinimize

Since the objective is not otherwise accessed, it does not need to be stored in the variable `obj` .

Adding constraints to the model is just as easy. For example, the constraint

```
-x[0] + x[1] + x[2] <= 20.0
```

can be added by calling:

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

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

Solve the model

Java methods create the object-oriented optimizer in a Java application of CPLEX.

So far you have seen some methods of IloCplex for creating models. All such methods are defined in the interfaces IloModeler and its extension IloMPModeler and IloCplexModeler. However, IloCplex not only implements these interfaces but also provides additional methods for solving a model and querying its results.

After you have created a model as explained in “Create the model” on page 68, the object IloCplex is ready to solve the problem, which consists of the model and all the modeling objects that have been added to it. Invoking the optimizer then is as simple as calling the method solve .

That method returns a Boolean value indicating whether the optimization succeeded in finding a solution. If no solution was found, `false` is returned. If `true` is returned, then CPLEX found a feasible solution, though it is not necessarily an optimal solution. More precise information about the outcome of the last call to the method `solve` can be obtained from the method getStatus .

The returned value tells you what CPLEX found out about the model: whether it found the optimal solution or only a feasible solution, whether it proved the model to be unbounded or infeasible, or whether nothing at all has been proved at this point. Even more detailed information about the termination of the optimizer call is available through the method getCplexStatus .

Query the results

Java methods query results from a Java application of CPLEX.

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

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

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

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

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

Building and solving a small LP model in Java

An example shows how to solve a model in a Java application of CPLEX.

Example: LPex1.java

This example illustrates solving a model in a Java application of CPLEX.

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

- “Modeling by rows” on page 72;
- “Modeling by columns” on page 72;
- “Modeling by nonzeros” on page 73.

Example `LPex1.java` is an extension of the example presented in “Entering the example” on page 21:

Maximize $x_1 + 2x_2 + 3x_3$

subject to

$$-x_1 + x_2 . + x_3 \leq 20$$

$$x_1 - 3x_2 + x_3 \leq 30$$

with these bounds

$$0 \leq x_1 \leq 40$$

$$0 \leq x_2 \leq infinity$$

$$0 \leq x_3 \leq infinity$$

After an initial check that a valid option string was provided as a calling argument, the program begins by enclosing all executable statements that follow in a `try/catch` pair of statements. In case of an error CPLEX Concert Technology will throw an exception of type IloException, which the `catch` statement then processes. In this simple example, an exception triggers the printing of a line stating `Concert exception 'e' caught` , where `e` is the specific exception.

First, create the model object `cplex` by executing the following statement:

```
IloCplex cplex = new IloCplex();
```

At this point, the `cplex` object represents an empty model, that is, a model with no variables, constraints, or other content. The model is then populated in one of several ways depending on the command line argument. The possible choices are implemented in the methods

- `populateByRow`
- `populateByColumn`
- `populateByNonzero`

All these methods pass the same three arguments. The first argument is the `cplex` object to be populated. The second and third arguments correspond to the variables (`var`) and range constraints (`rng`) respectively; the methods will write to `var[0]` and `rng[0]` an array of all the variables and constraints in the model, for later access.

After the model has been created in the `cplex` object, it is ready to be solved by a call to `cplex.solve`. The solution log will be output to the screen; this is because `IloCplex` prints all logging information to the `OutputStream` `cplex`. output, which by default is initialized to `System.out`. You can change this by calling the method `cplex`. setOut. In particular, you can turn off logging by setting the output stream to `null`, that is, by calling `cplex.setOut(null)`. Similarly, `IloCplex` issues warning messages to `cplex`. warning, and `cplex`. setWarning can be used to change (or turn off) the `OutputStream` that will be used.

If the `solve` method finds a feasible solution for the active model, it returns `true`. The next section of code accesses the solution. The method `cplex`. getValues(`var[0]`) returns an array of primal solution values for all the variables. This array is stored as `double[]x` . The values in `x` are ordered such that `x[j]` is the primal solution value for variable `var[0][j]` . Similarly, the reduced costs, duals, and slack values are queried and stored in arrays `dj`, `pi`, and `slack`, respectively. Finally, the solution status of the active model and the objective value of the solution are queried with the methods `IloCplex`. getStatus and `IloCplex`. getObjValue, respectively. The program then concludes by printing the values that have been obtained in the previous steps, and terminates after calling `cplex`. end to

free the memory used by the model object; the `catch` method of IloException provides screen output in case of any error conditions along the way.

The remainder of the example source code is devoted to the details of populating the model object and the following three sections provide details on how the methods work.

You can view the complete program online in the standard distribution of the product at *yourCPLEXinstallation* `/examples/src/LPex1.java`.

Modeling by rows

Java methods support modeling by rows in this example of a Java application of CPLEX.

The method `populateByRow` creates the model by adding the finished constraints and objective function to the active model, one by one. It does so by first creating the variables with the method cplex. numVarArray. Then the minimization objective function is created and added to the active model with the method `IloCplex`. addMinimize. The expression that defines the objective function is created by a method, `IloCplex`. scalProd, that forms a scalar product using an array of objective coefficients times the array of variables. Finally, each of the two constraints of the model are created and added to the active model with the method IloCplex. addLe. For building the constraint expression, the methods `IloCplex`. sum and `IloCplex`. prod are used, as a contrast to the approach used in constructing the objective function.

Modeling by columns

Java methods support modeling by columns in this example of a Java application of CPLEX.

While for many examples population by rows may seem most straightforward and natural, there are some models where population by columns is a more natural or more efficient approach to implement. For example, problems with network structure typically lend themselves well to modeling by column. Readers familiar with matrix algebra may view the method `populateByColumn` as producing the transpose of what is produced by the method `populateByRow` . In contrast to modeling by rows, modeling by columns means that the coefficients of the constraint matrix are given in a column-wise way. As each column represents the constraint coefficients for a given variable in the linear program, this modeling approach is most natural where it is easy to access the matrix coefficients by iterating through all the variables, such as in network flow problems.

Range objects are created for modeling by column with only their lower and upper bound. No expressions are given; building them at this point would be impossible since the variables have not been created yet. Similarly, the objective function is created only with its intended optimization sense, and without any expression.

Next the variables are created and installed in the existing ranges and objective. These newly created variables are introduced into the ranges and the objective by means of column objects, which are implemented in the class IloColumn. Objects of this class are created with the methods IloCplex. column, and can be linked together with the method IloColumn. and to form aggregate IloColumn objects.

An instance of IloColumn created with the method IloCplex. column contains information about how to use this column to introduce a new variable into an

existing modeling object. For example, if obj is an instance of a class that implements the interface IloObjective, then cplex.column(obj, 2.0) creates an instance of IloColumn containing the information to install a new variable in the expression of the IloObjective object obj with a linear coefficient of 2.0. Similarly, for rng, a constraint that is an instance of a class that implements the interface IloRange, the invocation of the method cplex.column(rng, -1.0) creates an IloColumn object containing the information to install a new variable into the expression of rng , as a linear term with coefficient -1.0.

When you use the approach of modeling by column, new columns are created and installed as variables in all existing modeling objects where they are needed. To do this with Concert Technology, you create an IloColumn object for every modeling object in which you want to install a new variable, and link them together with the method IloColumn. and . For example, the first variable in populateByColumn is created like this:

The three methods model.column create IloColumn objects for installing a new variable in the objective obj and in the constraints r0 and r1, with linear coefficients 1.0 , -1.0, and 1.0, respectively. They are all linked to an aggregate column object by the method and. This aggregate column object is passed as the first argument to the method numVar, along with the bounds 0.0 and 40.0 as the other two arguments. The method numVar now creates a new variable and immediately installs it in the modeling objects obj, r0, and r1 as defined by the aggregate column object. After it has been installed, the new variable is returned and stored in var[0][0].

Modeling by nonzeros

Java methods support modeling by nonzeros in this example of a Java application of CPLEX.

The last of the three functions for building the model is populateByNonzero . This function creates the variables with only their bounds, and the empty constraints, that is, ranged constraints with only lower and upper bound but with no expression. Only after that are the expressions constructed over these existing variables, in a manner similar to the ones already described; they are installed in the existing constraints with the method IloRange. setExpr .

Chapter 6. Concert Technology tutorial for .NET users

CPLEX applications written in C#.NET use Concert Technology in the .NET framework.

Presenting the tutorial

CPLEX in the .NET framework supports creating a model, populating it with data, solving the problem, and displaying results of the solution.

This tutorial introduces CPLEX through Concert Technology in the .NET framework. It gives you an overview of a typical application and highlights procedures for:
- Creating a model
- Populating the model with data, either by rows, by columns, or by nonzeros
- Solving that model
- Displaying results after solving

This chapter concentrates on an example using C#.NET. There are also examples of VB.NET (Visual Basic in the .NET framework) delivered with CPLEX in *yourCPLEXhome*`\examples\src\vb` . Because of their .NET framework, those VB.NET examples differ from the traditional Visual Basic examples that may already be familiar to some CPLEX users.

In the standard distribution of the product, the file `dotnet.html` offers useful details about installing the product as well as compiling and executing examples.

Note:

This tutorial is based on a procedure-based learning strategy. The tutorial is built around a sample problem, available in a file that can be opened in an integrated development environment, such as Microsoft Visual Studio. As you follow the steps in the tutorial, you can examine the code and apply concepts explained in the tutorial. Then you compile and execute the code to analyze the results. Ideally, as you work through the tutorial, you are sitting in front of your computer with CPLEX and Concert Technology for .NET users already installed and available in Microsoft Visual Studio.

What you need to know: prerequisites

Prerequisites for this tutorial include knowledge of C#.NET, familiarity with linking, compiling, and executing, as well as a quick test of your installation of CPLEX.

This tutorial requires a working knowledge of C#.NET.

If you are experienced in mathematical programming or operations research, you are probably already familiar with many concepts used in this tutorial. However, little or no experience in mathematical programming or operations research is required to follow this tutorial.

© Copyright IBM Corp. 1987, 2013

You should have CPLEX and Concert Technology for .NET users **installed** in your development environment before starting this tutorial. In your integrated development environment, you should be able to **compile**, **link**, and **execute** a sample application provided with CPLEX and Concert Technology for .NET users before starting the tutorial.

To check your installation before starting the tutorial, open

yourCPLEXhome `\examples\`*platform*`\`*format*`\examples.net.sln`

in your integrated development environment, where *yourCPLEXhome* specifies the place you installed CPLEX on your platform, and *format* specifies one of these possibilities: `stat_mda`, `stat_mta`, or `stat_sta`. Your integrated development environment, Microsoft Visual Studio, will then check for the DLLs of CPLEX and Concert Technology for .NET users and warn you if they are not available to it.

Another way to check your installation is to load the project for one of the samples delivered with your product. For example, you might load the following project into Microsoft Visual Studio to check a C# example of the diet problem:

yourCPLEXhome`\examples\`*platform*`\`*format*`\Diet.csproj`

What you will be doing

This tutorial walks you through building and solving a small linear programming model in C#.NET..

CPLEX can work together with Concert Technology for .NET users, a .NET library that allows you to model optimization problems independently of the algorithms used to solve the problem. It provides an extensible modeling layer adapted to a variety of algorithms ready to use off the shelf. This modeling layer enables you to change your model, without completely rewriting your application.

To find a solution to a problem by means of CPLEX with Concert Technology for .NET users, you use a three-stage method: describe, model, and solve.

The first stage is to describe the problem in natural language.

The second stage is to use the classes and interfaces of Concert Technology for .NET users to model the problem. The model is composed of data, decision variables, and constraints. Decision variables are the unknown information in a problem. Each decision variable has a domain of possible values. The constraints are limits or restrictions on combinations of values for these decision variables. The model may also contain an objective, an expression that can be maximized or minimized.

The third stage is to use the classes of Concert Technology for .NET users to solve the problem. Solving the problem consists of finding a value for each decision variable while simultaneously satisfying the constraints and maximizing or minimizing an objective, if one is included in the model.

In these tutorials, you will describe, model, and solve a simple problem that also appears elsewhere in C, C++, and Java chapters of this manual:

- “Building and solving a small LP model in C” on page 90
- “Building and solving a small LP model in C++” on page 55

- “Building and solving a small LP model in Java” on page 70

Describe

The first step is for you to describe the problem in natural language and answer basic questions about the problem.

- What is the known information in this problem? That is, what data is available?
- What is the unknown information in this problem? That is, what are the decision variables?
- What are the limitations in the problem? That is, what are the constraints on the decision variables?
- What is the purpose of solving this problem? That is, what is the objective function?

Note:

Though the **Describe** step of the process may seem trivial in a simple problem like this one, you will find that taking the time to fully describe a more complex problem is vital for creating a successful application. You will be able to code your application more quickly and effectively if you take the time to describe the model, isolating the decision variables, constraints, and objective.

Model

The second stage is for you to use the classes of Concert Technology for .NET users to build a model of the problem. The model is composed of *decision variables* and *constraints* on those variables. The model of this problem also contains an *objective*.

Solve

The third stage is for you to use an instance of the class `Cplex` to search for a solution and to solve the problem. Solving the problem consists of finding a value for each variable while simultaneously satisfying the constraints and minimizing the objective.

Describe

Ask these questions to describe an optimization problem adequately for application development.

The aim in this tutorial is to see three different ways to build a model: by rows, by columns, or by nonzeros. After building the model of the problem in one of those ways, the application optimizes the problem and displays the solution.

Step One: Describe the problem

Write a natural language description of the problem and answer these questions:

- What is known about the problem?
- What are the unknown pieces of information (the decision variables) in this problem?
- What are the limitations (the constraints) on the decision variables?
- What is the purpose (the objective) of solving this problem?

Building a small LP problem in C#

Here is a conventional formulation of the problem that the example optimizes:

Maximize $x_1 + 2x_2 + 3x_3$

subject to

$-x_1 + x_2 + x_3 \leq 20$

$x_1 - 3x_2 + x_3 \leq 30$

with these bounds

$0 \leq x_1 \leq 40$

$0 \leq x_2 \leq infinity$

$0 \leq x_3 \leq infinity$

- What are the decision variables in this problem?

  x_1, x_2, x_3

- What are the constraints?

  $-x_1 + x_2 + x_3 \leq 20$

  $x_1 - 3x_2 + x_3 \leq 30$

  $0 \leq x_1 \leq 40$

  $0 \leq x_2 \leq \text{infinity}$

  $0 \leq x_3 \leq \text{infinity}$

- What is the objective?

  Maximize $x_1 + 2x_2 + 3x_3$

Model

Use classes of Concert Technology for .NET users to build a model for the problem.

After you have written a description of the problem, you can use classes of Concert Technology for .NET users with CPLEX to build a model.

Step 2: Open the file

Open the file *yourCPLEXhome* `\examples\src\tutorials\LPex1lesson.cs` in your integrated development environment, such as Microsoft Visual Studio.

Step 3: Create the model object

Go to the comment **Step 3** in that file, and add this statement to create the `Cplex` model for your application.

```
Cplex cplex = new Cplex();
```

That statement creates an empty instance of the class `Cplex` . In the next steps, you will add methods that make it possible for your application populate the model with data, either by rows, by columns, or by nonzeros.

Step 4: Populate the model by rows

Now go to the comment **Step 4** in that file, and add these lines to create a method to populate the empty model with data by rows.

```
internal static void PopulateByRow(IMPModeler model,
                                   INumVar[][] var,
                                   IRange[][] rng) {
  double[] lb = {0.0, 0.0, 0.0};
  double[] ub = {40.0,
                 System.Double.MaxValue,
                 System.Double.MaxValue};
  INumVar[] x  = model.NumVarArray(3, lb, ub);
  var[0] = x;

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

  rng[0] = new IRange[2];
  rng[0][0] = model.AddLe(model.Sum(model.Prod(-1.0, x[0]),
                                    model.Prod( 1.0, x[1]),
                                    model.Prod( 1.0, x[2])), 20.0);
  rng[0][1] = model.AddLe(model.Sum(model.Prod( 1.0, x[0]),
                                    model.Prod(-3.0, x[1]),
                                    model.Prod( 1.0, x[2])), 30.0);
}
```

Those lines populate the model with data specific to this particular example. However, you can see from its use of the interface `IMPModeler` how to add *ranged constraint*s to a model. `IMPModeler` is the Concert Technology interface typically used to build math programming (MP) matrix models. You will see its use again in Step 5 and Step 6.

Step 5: Populate the model by columns

Go to the comment **Step 5** in the file, and add these lines to create a method to populate the empty model with data by columns.

```
internal static void PopulateByColumn(IMPModeler model,
                             INumVar[][] var,
                             IRange[][] rng) {
   IObjective obj = model.AddMaximize();

   rng[0] = new IRange[2];
   rng[0][0] = model.AddRange(-System.Double.MaxValue, 20.0);
   rng[0][1] = model.AddRange(-System.Double.MaxValue, 30.0);

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

   var[0] = new INumVar[3];
   var[0][0] = model.NumVar(model.Column(obj,  1.0).And(
                            model.Column(r0,  -1.0).And(
                            model.Column(r1,   1.0))),
                            0.0, 40.0);
   var[0][1] = model.NumVar(model.Column(obj,  2.0).And(
                            model.Column(r0,   1.0).And(
                            model.Column(r1,  -3.0))),
                            0.0, System.Double.MaxValue);
   var[0][2] = model.NumVar(model.Column(obj,  3.0).And(
                            model.Column(r0,   1.0).And(
                            model.Column(r1,   1.0))),
                            0.0, System.Double.MaxValue);
}
```

Again, those lines populate the model with data specific to this problem. From them you can see how to use the interface IMPModeler to add *columns* to an empty model.

While for many examples population by rows may seem most straightforward and natural, there are some models where population by columns is a more natural or more efficient approach to implement. For example, problems with network structure typically lend themselves well to modeling by column. Readers familiar with matrix algebra may view the method populateByColumn as the transpose of populateByRow .

In this approach, range objects are created for modeling by column with only their lower and upper bound. No *expressions* over variables are given because building them at this point would be impossible since the variables have not been created yet. Similarly, the objective function is created only with its intended optimization sense, and without any expression.

Next the variables are created and installed in the existing ranges and objective. These newly created variables are introduced into the ranges and the objective by means of column objects, which are implemented in the class IColumn. Objects of this class are created with the methods Cplex.Column, and can be linked together with the method IColumn.And to form aggregate IColumn objects.

An IColumn object created with the method ICplex.Column contains information about how to use this column to introduce a new variable into an existing modeling object. For example if obj is an IObjective object, cplex.Column(obj, 2.0) creates an IColumn object containing the information to install a new variable in the expression of the IObjective object obj with a linear coefficient of 2.0 . Similarly, for an IRange constraint rng , the method call cplex.Column(rng, -1.0) creates an IColumn object containing the information to install a new variable into the expression of rng , as a linear term with coefficient -1.0 .

In short, when you use a modeling-by-column approach, new columns are created and installed as variables in all existing modeling objects where they are needed. To do this with Concert Technology, you create an IColumn object for every modeling object in which you want to install a new variable, and link them together with the method IColumn.And .

Step 6: Populate the model by nonzeros

Go to the comment **Step 6** in the file, and add these lines to create a method to populate the empty model with data by nonzeros.

```
internal static void PopulateByNonzero(IMPModeler model,
                             INumVar[][] var,
                             IRange[][] rng) {
  double[]    lb = {0.0, 0.0, 0.0};
  double[]    ub = {40.0, System.Double.MaxValue, System.Double.MaxValue};
  INumVar[] x  = model.NumVarArray(3, lb, ub);
  var[0] = x;

  double[] objvals = {1.0, 2.0, 3.0};
  model.Add(model.Maximize(model.ScalProd(x, objvals)));

  rng[0] = new IRange[2];
  rng[0][0] = model.AddRange(-System.Double.MaxValue, 20.0);
  rng[0][1] = model.AddRange(-System.Double.MaxValue, 30.0);

  rng[0][0].Expr = model.Sum(model.Prod(-1.0, x[0]),
```

```
                                model.Prod( 1.0, x[1]),
                                model.Prod( 1.0, x[2]));
      rng[0][1].Expr = model.Sum(model.Prod( 1.0, x[0]),
                                model.Prod(-3.0, x[1]),
                                model.Prod( 1.0, x[2]));
   }
```

In those lines, you can see how to populate an empty model with data indicating the nonzeros of the constraint matrix. Those lines first create objects for the objective and the ranges without expressions. They also create variables without columns; that is, variables with only their bounds. Then those lines create *expressions* over the objective, ranges, and variables and add the expressions to the model.

Step 7: Add an interface

Go to the comment **Step 7** in the file, and add these lines to create a method that tells a user how to invoke this application.

```
 internal static void Usage() {
    System.Console.WriteLine("usage:   LPex1 <option>");
    System.Console.WriteLine("options: -r   build model row by row");
    System.Console.WriteLine("options: -c   build model column by column");
    System.Console.WriteLine("options: -n   build model nonzero by nonzero");
 }
```

Step 8: Add a command evaluator

Go to the comment **Step 8** in the file, and add these lines to create a switch statement that evaluates the command that a user of your application might enter.

```
switch ( args[0].ToCharArray()[1] ) {
case 'r': PopulateByRow(cplex, var, rng);
          break;
case 'c': PopulateByColumn(cplex, var, rng);
          break;
case 'n': PopulateByNonzero(cplex, var, rng);
          break;
default:  Usage();
          return;
}
```

Solve

Add the parts of the application that solve the problem.

After you have declared the decision variables and added the constraints and objective function to the model, your application is ready to search for a solution.

Step 9: Search for a solution

Go to **Step 9** in the file, and add this line to make your application search for a solution.

```
if ( cplex.Solve() ) {
```

Step 10: Display the solution

Go to the comment **Step 10** in the file, and add these lines to enable your application to display any solution found in Step 9.

```
            double[] x     = cplex.GetValues(var[0]);
            double[] dj    = cplex.GetReducedCosts(var[0]);
            double[] pi    = cplex.GetDuals(rng[0]);
            double[] slack = cplex.GetSlacks(rng[0]);

            cplex.Output().WriteLine("Solution status = "
                                     + cplex.GetStatus());
            cplex.Output().WriteLine("Solution value  = "
                                     + cplex.ObjValue);

            int nvars = x.Length;
            for (int j = 0; j < nvars; ++j) {
               cplex.Output().WriteLine("Variable  :"
                                         + j
                                         +" Value = "
                                         + x[j]
                                         +" Reduced cost = "
                                         + dj[j]);
            }

            int ncons = slack.Length;
            for (int i = 0; i < ncons; ++i) {
               cplex.Output().WriteLine("Constraint:"
                                         + i
                                         +" Slack = "
                                         + slack[i]
                                         +" Pi = "
                                         + pi[i]);
            }
         }
```

Step 11: Save the model to a file

If you want to save your model to a file in LP format, go to the comment **Step 11** in your application file, and add this line.

```
cplex.ExportModel("lpex1.lp");
```

If you have followed the steps in this tutorial interactively, you now have a complete application that you can compile and execute.

Complete program

A copy of the sample is available on line.

You can view the complete program online in the standard distribution of the product at *yourCPLEXinstallation* `\examples\src\csharp\LPex1.cs` .

Chapter 7. Callable Library tutorial

Applications written in C use the CPLEX Callable Library (C API).

The design of the CPLEX Callable Library

The architecture of the Callable Library (C API) supports user-written applications in C and other programming languages callable from C.

Figure 5 shows a picture of the IBM ILOG CPLEX world. The CPLEX Callable Library together with the CPLEX internals make up the CPLEX core. The core becomes associated with your application through Callable Library routines. The CPLEX environment and all problem-defining data are established inside the CPLEX core.

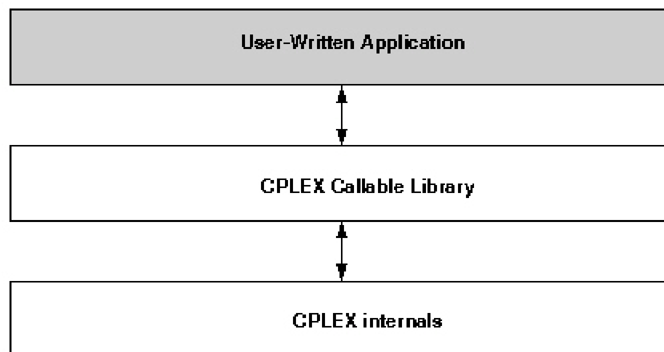

Figure 5. A View of the CPLEX Callable Library

The CPLEX Callable Library includes several categories of routines:

- optimization and result routines for defining a problem, optimizing it, and getting the results;
- utility routines for addressing application programming matters;
- problem modification routines to change a problem after it has been created within the CPLEX internals;
- problem query routines to access information about a problem after it has been created;
- file reading and writing routines to move information from the file system into your application or out of your application to the file system;
- parameter setting and query routines to access and modify the values of control parameters maintained by CPLEX.

Compiling and linking Callable Library applications

Compilation and linking an application of the C API differ according to platform.

Overview

Callable Library applications follow conventional coding practices for linking and compiling.

© Copyright IBM Corp. 1987, 2013

Each Callable Library is distributed as a single library file `libcplex.a` or `cplexXXX.lib` . Use of the library file is similar to that with `.o` or `.obj` files. Simply substitute the library file in the link procedure. This procedure simplifies linking and makes sure that the smallest possible executable is generated.

The following compilation and linking instructions assume that the example source programs and CPLEX Callable Library files are in the directories associated with a default installation of the software. If this is not true, additional compile and link flags would be required to point to the locations of the include file `cplex.h` or `cplexx.h`, and Callable Library files respectively.

Note:

The instructions below were current at the time of publication. As compilers, linkers and operating systems are released, different instructions may apply. Be sure to check the *Release Notes* that come with your CPLEX distribution for any changes. Also check the CPLEX topics among Detailed System Requirements (DSR) at the IBM.com website.

Building Callable Library applications on UNIX platforms

Standard conventions for compiling and linking on UNIX platforms apply to your Callable Library applications.

To compile and execute an example (`lpex1`) do the following tasks:

```
  % cd examples/platform/format
% make lpex1 #to compile and execute the first CPLEX example
```

In that command, *platform* specifies the name of the subdirectory corresponding to your type of machine, and *format* specifies your particular library format, such as static, multithreaded, or other format.

A list of all the examples that can be built this way is to be found in the `makefile` by looking for `C_EX` (C examples), or you can view the files listed in `examples/src`.

The `makefile` contains recommended compiler flags and other settings for your particular computer, which you can find by searching in it for "Compiler options" and use in your applications that call CPLEX.

Building Callable Library applications on Win32 platforms

Standard conventions for compiling and linking on various Windows platforms apply to your Callable Library applications.

Building a CPLEX application using Microsoft Visual C++ Integrated Development Environment, or the Microsoft Visual C++ command line compiler are explained here.

Microsoft Visual C++ IDE

To make a CPLEX Callable Library application using Visual C++, first create or open a project in the Visual C++ Integrated Development Environment (IDE). Project files are provided for each of the examples found in the directory or folder `examples\`*platform**format* where *platform* and *format* refer to your type of machine and compiler. For details about the build process, refer to the information file `msvc.html`, which is found in the top of the installed CPLEX directory structure.

Note:

The distributed application must be able to locate `CPLEXXXX.dll` at run time.

Microsoft Visual C++ Command Line Compiler

If the Visual C++ command line compiler is used outside of the IDE, the command should resemble the following example. The example command assumes that the file `cplex XXX .lib` is in the current directory with the source file `lpex1.c` , and that the line in the source file "`#include <ilcplex/cplex.h>` " correctly points to the location of the include file or else has been modified to do so (or that the directories containing these files have been added to the environment variables `LIB` and `INCLUDE` respectively).

```
cl lpex1.c cplexXXX.lib
```

This command will create the executable file `lpex1.exe` .

Using Dynamic Loading

Some projects require more precise control over the loading and unloading of DLLs. For information on loading and unloading DLLs without using static linking, please refer to the compiler documentation or to a book such as *Advanced Windows* by Jeffrey Richter from Microsoft Press. If this is not a requirement, the static link implementations already mentioned are easier to use.

How CPLEX works

CPLEX carries out these activities when it is invoked from the Callable Library (C API).

Overview

Your C code includes these basic steps in a Callable Library application.

When your application uses routines of the CPLEX Callable Library, it must first open the CPLEX environment, then create and populate a problem object before it solves a problem. Before it exits, the application must also free the problem object and release the CPLEX environment. The following sections explain those steps.

Opening the CPLEX environment

Your Callable Library application initializes the CPLEX environment.

CPLEX requires a number of internal data structures in order to execute properly. These data structures must be initialized before any call to the CPLEX Callable Library. The first call to the CPLEX Callable Library is always to the routine `CPXopenCPLEX`. This routine returns a pointer to the CPLEX environment. This pointer is then passed to every CPLEX Callable Library routine, except those, such as `CPXmsg`, which do not require an environment.

The application developer must make an independent decision as to whether the variable containing the environment pointer is a global or local variable. Multiple environments are allowed, but extensive opening and closing of environments may create significant overhead and degrade performance; typical applications make use of only one environment for the entire execution, since a single environment may hold as many problem objects as the user wants. After all calls to the Callable Library are complete, the environment is released by the routine `CPXcloseCPLEX`.

This routine specifies to CPLEX that all calls to the Callable Library are complete, any memory allocated by CPLEX is returned to the operating system, and the use of CPLEX is ended for this run.

Instantiating the problem object

Your Callable Library application creates a problem object in the environment.

A *problem object* is instantiated (created and initialized) by CPLEX when you call the routine `CPXcreateprob`. It is destroyed when you call `CPXfreeprob`. CPLEX allows you to create more than one problem object, although typical applications will use only one. Each problem object is referenced by a pointer returned by `CPXcreateprob` and represents one specific problem instance. Most Callable Library functions (except parameter setting functions and message handling functions) require a pointer to a problem object.

Note:

An attempt to use a problem object in any environment other than the environment (or a child of that environment) where the problem object was created will raise an error.

Populating the problem object

Your Callable Library application populates the problem object with data.

The problem object instantiated by `CPXcreateprob` represents an empty problem that contains no data; it has zero constraints, zero variables, and an empty constraint matrix. This empty problem object must be populated with data. This step can be carried out in several ways.

- The problem object can be populated by assembling arrays of data and then calling `CPXcopylp` to copy the data into the problem object. (For example, see "Building and solving a small LP model in C" on page 90.)
- Alternatively, you can populate the problem object by sequences of calls to the routines `CPXnewcols`, `CPXnewrows`, `CPXaddcols`, `CPXaddrows`, and `CPXchgcoeflist` ; these routines may be called in any order that is convenient. (For example, see "Adding rows to a problem: example lpex3.c" on page 92.)
- If the data already exist in a file using MPS format or LP format, you can use `CPXreadcopyprob` to read the file and copy the data into the problem object. (For example, see "Reading a problem from a file: example lpex2.c" on page 91.)

Changing the problem object

If your Callable Library application modifies the problem object, consider these issues.

A major consideration in the design of CPLEX is the need to re-optimize modified linear programs efficiently. In order to accomplish that, CPLEX must be aware of changes that have been made to a linear program since it was last optimized. Problem modification routines are available in the Callable Library.

Do not change the problem by changing the original problem data arrays and then making a call to `CPXcopylp`. Instead, change the problem using the problem modification routines, allowing CPLEX to make use of as much solution information as possible from the solution of the problem before the modifications took place.

For example, suppose that a problem has been solved, and that the user has changed the upper bound on a variable through an appropriate call to the CPLEX Callable Library. A re-optimization would then begin from the previous optimal basis, and if that old basis were still optimal, then that information would be returned without even the need to refactor the old basis.

Creating a successful Callable Library application

Successful applications of the C API often follow these guidelines.

Overview

This outline suggests useful steps in developing a Callable Library application.

Callable Library applications are created to solve a wide variety of problems. Each application shares certain common characteristics, regardless of its apparent uniqueness. The following steps can help you minimize development time and get maximum performance from your programs:

Prototype the model

A small model for prototyping may be helpful, especially in applications to solve very large models.

Create a small model of the problem to be solved. An algebraic modeling language is sometimes helpful during this step.

Identify the routines to call

This guideline may help you identify relevant Callable Library routines to use.

By separating the application into smaller components, you can easily identify the tools needed to complete the application. Part of this process consists of identifying the Callable Library routines that will be called by your application.

In some applications, the Callable Library is a small part of a larger program. In that case, the only CPLEX routines needed may be for:

- problem creation;
- optimizing;
- obtaining results.

In other cases the Callable Library is used extensively in the application. If so, Callable Library routines may also be needed to:

- modify the problem;
- set parameters;
- manage input and output messages and files;
- query problem data.

Test procedures in the application

The Interactive Optimizer offers a test-bed for Callable Library applications.

It is often possible to test the procedures of an application in the CPLEX Interactive Optimizer with a small prototype of the model. Doing so will help identify the Callable Library routines required. The test may also uncover any flaws in procedure logic before you invest significant development effort.

Trying the CPLEX Interactive Optimizer is an easy way to decide the best optimization procedure and parameter settings.

Assemble the data

To populate the model with data, CPLEX offers alternative routines.

You must decide which approach to populating the problem object is best for your application. Reading an MPS or LP file may reduce the coding effort but can increase the run-time and disk-space requirements of the program. Building the problem in memory and then calling `CPXcopylp` avoids time consuming disk-file reading. Using the routines `CPXnewcols`, `CPXnewrows`, `CPXaddcols`, `CPXaddrows`, and `CPXchgcoeflist` can lead to modular code that may be more easily maintained than if you assemble all model data in one step.

Another consideration is that if the Callable Library application reads an MPS or LP formatted file, usually another application is required to generate that file. Particularly in the case of MPS files, the data structures used to generate the file could almost certainly be used to build the problem-defining arrays for `CPXcopylp` directly. The result would be less coding and a faster, more efficient application. These observations suggest that formatted files may be useful when prototyping your application, while assembling the arrays in memory may be a useful enhancement for a production application.

Choose an optimizer

According to problem type, consider which optimizer to use.

After a problem object has been instantiated and populated, it can be solved using one of the optimizers provided by the CPLEX Callable Library. The choice of optimizer depends on the problem type.

- LP and QP problems can be solved by:
  - the primal simplex optimizer;
  - the dual simplex optimizer; and
  - the barrier optimizer.
- LP and QP problems with a substantial network can also be solved by a special network optimizer.
- LP problems can also be solved by:
  - the sifting optimizer; and
  - the concurrent optimizer.
- If the problem includes integer variables, mixed integer programming (MIP) must be used.

There are also many different possible parameter settings for each optimizer. The default values will usually be the best for linear programs. Integer programming problems are more sensitive to specific settings, so additional experimentation will often be useful.

Choosing the best way to solve the problem can dramatically improve performance. For more information, refer to the sections about tuning LP performance and trouble-shooting MIP performance in the *CPLEX User's Manual*.

Observe good programming practices

CPLEX supports standard programming conventions in Callable Library applications.

Using good programming practices will save development time and make the program easier to understand and modify. A list of good programming practices is provided in the *CPLEX User's Manual*, in Developing CPLEX applications.

Debug your program

CPLEX is compatible with debuggers and other programming tools.

Your program may not run properly the first time you build it. Learn to use a symbolic debugger and other widely available tools that facilitate the creation of error-free code. Use the list of debugging tips provided in the *CPLEX User's Manual* to find and correct problems in your Callable Library application.

Test your application

CPLEX supports testing of performance and correctness.

After an application works correctly, it still may have errors or features that inhibit execution speed. To get the most out of your application, be sure to test its performance as well as its correctness. Again, the Interactive Optimizer can help. Since the Interactive Optimizer uses the same routines as the Callable Library, it should take the same amount of time to solve a problem as a Callable Library application.

Use the `CPXwriteprob` routine with the SAV format to create a binary representation of the problem object, then read it in and solve it with the Interactive Optimizer. If the application sets optimization parameters, use the same settings with the Interactive Optimizer. If your application takes significantly longer than the Interactive Optimizer, performance within your application can probably be improved. In such a case, possible performance inhibitors include fragmentation of memory, unnecessary compiler and linker options, and coding approaches that slow the program without causing it to give incorrect results.

Use the examples

CPLEX offers examples to follow.

The CPLEX Callable Library is distributed with a variety of examples that illustrate the flexibility of the Callable Library. The C source of all examples is provided in the standard distribution. For explanations about the examples of quadratic programming problems (QPs), mixed integer programming problems (MIPs) and network flows, see the *CPLEX User's Manual*. Explanations of the following examples of LPs appear in this manual:

`lpex1.c`
: illustrates various ways of generating a problem object.

`lpex2.c`
: demonstrates how to read a problem from a file, optimize it via a choice of several means, and obtain the solution.

`lpex3.c`
: demonstrates how to add rows to a problem object and re-optimize.

It is a good idea to compile, link, and run all of the examples provided in the standard distribution.

Building and solving a small LP model in C

This application in the C API introduces basic features of the Callable Library.

The example `lpex1.c` shows you how to use problem modification routines from the CPLEX Callable Library in three different ways to build a model. The application in the example takes a single command line argument that indicates whether to build the constraint matrix by rows, columns, or nonzeros. After building the problem, the application optimizes it and displays the solution. Here is the problem that the example optimizes:

Maximize	$x_1 + 2x_2 + 3x_3$
subject to	
	$-x_1 + x_2 + x_3 \le 20$
	$x_1 - 3x_2 + x_3 \le 30$
with these bounds	
	$0 \le x_1 \le 40$
	$0 \le x_2 \le infinity$
	$0 \le x_3 \le infinity$

Before any CPLEX Callable Library routine can be called, your application must call the routine `CPXopenCPLEX` to get a pointer (called `env`) to the CPLEX environment. Your application will then pass this pointer to every Callable Library routine. If this routine fails, it returns an error code. This error code can be translated to a string by the routine `CPXgeterrorstring`.

After the CPLEX environment is initialized, the CPLEX screen switch parameter (`CPX_PARAM_SCRIND`) is turned on by the routine `CPXsetintparam` . This causes all default CPLEX output to appear on the screen. If this parameter is not set, then CPLEX will generate no viewable output on the screen or in a file.

At this point, the routine `CPXcreateprob` is called to create an empty problem object. Based on the problem-building method selected by the command-line argument, the application then calls a routine to build the matrix by rows, by columns, or by nonzeros. The routine `populatebyrow` first calls `CPXnewcols` to specify the column-based problem data, such as the objective, bounds, and variables names. The routine `CPXaddrows` is then called to supply the constraints. The routine `populatebycolumn` first calls `CPXnewrows` to specify the row-based problem data, such as the righthand side values and sense of constraints. The routine `CPXaddcols` is then called to supply the columns of the matrix and the associated column bounds, names, and objective coefficients. The routine `populatebynonzero` calls both `CPXnewrows` and `CPXnewcols` to supply all the problem data except the actual constraint matrix. At this point, the rows and columns are well defined, but the constraint matrix remains empty. The routine `CPXchgcoeflist` is then called to fill in the nonzero entries in the matrix.

After the problem has been specified, the application optimizes it by calling the routine `CPXlpopt`. Its default behavior is to use the CPLEX dual simplex optimizer. If this routine returns a nonzero result, then an error occurred. If no error occurred, the application allocates arrays for solution values of the primal variables, dual variables, slack variables, and reduced costs; then it obtains the solution

information by calling the routine `CPXsolution`. This routine returns the status of the problem (whether optimal, infeasible, or unbounded, and whether a time limit or iteration limit was reached), the objective value and the solution vectors. The application then displays this information on the screen.

As a debugging aid, the application writes the problem to a CPLEX LP file (named `lpex1.lp`) by calling the routine `CPXwriteprob`. This file can be examined to detect whether any errors occurred in the routines creating the problem. `CPXwriteprob` can be called at any time after `CPXcreateprob` has created the `lp` pointer.

The label `TERMINATE` : is used as a place for the program to exit if any type of failure occurs, or if everything succeeds. In either case, the problem object represented by `lp` is released by the call to `CPXfreeprob`, and any memory allocated for solution arrays is freed. The application then calls `CPXcloseCPLEX`; it tells CPLEX that all calls to the Callable Library are complete. If an error occurs when this routine is called, then a call to `CPXgeterrorstring` is needed to retrieve the error message, since `CPXcloseCPLEX` causes no screen output.

You can view the complete program online in the standard distribution of the product at *yourCPLEXinstallation*`/examples/src/lpex1.c` .

Reading a problem from a file: example lpex2.c

This application reads the model from a formatted file.

The previous example, `lpex1.c` , shows a way to copy problem data into a problem object as part of an application that calls routines from the CPLEX Callable Library. Frequently, however, a file already exists containing a linear programming problem in the industry standard MPS format, the CPLEX LP format, or the CPLEX binary SAV format. In example `lpex2.c` , CPLEX file-reading and optimization routines read such a file to solve the problem.

Example `lpex2.c` uses command line arguments to specify the name of the input file and the optimizer to call.

Usage: `lpex2 filename optimizer`

Where: `filename` is a file with extension MPS, SAV, or LP (lower case is allowed), and `optimizer` is one of the following letters:

o	default
p	primal simplex
d	dual simplex
n	network with dual simplex cleanup
h	barrier with crossover
b	barrier without crossover
s	sifting
c	concurrent

For example, this command:

```
lpex2 example.mps d
```

reads the file `example.mps` and solves the problem with the dual simplex optimizer.

To illustrate the ease of reading a problem, the example uses the routine `CPXreadcopyprob`. This routine detects the type of the file, reads the file, and copies the data into the CPLEX problem object that is created with a call to `CPXcreateprob`. The user need not be concerned with the memory management of the data. Memory management is handled transparently by `CPXreadcopyprob` .

After calling `CPXopenCPLEX` and turning on the screen switch by setting the `CPX_PARAM_SCRIND` parameter to `CPX_ON`, the example creates an empty problem object with a call to `CPXcreateprob`. This call returns a pointer, `lp` , to the new problem object. Then the data is read in by the routine `CPXreadcopyprob`. After the data is copied, the appropriate optimization routine is called, based on the command line argument.

After optimization, a call to `CPXgetstat` retrieves the status of the solution . The cases of infeasibility or unboundedness in the model are handled in a simple fashion here; a more complex application program might treat these cases in more detail. With these two cases out of the way, the program then calls `CPXsolninfo` to examine the nature of the solution. Certain that a solution exists, the application then calls `CPXgetobjval` to obtain the objective function value for this solution and report it.

Next, preparations are made to print the solution value and basis status of each individual variable, by allocating arrays of appropriate size; these sizes are detected by calls to the routines `CPXgetnumcols` and `CPXgetnumrows` . Note that a basis is not guaranteed to exist, depending on which optimizer was selected at run time, so some of these steps, including the call to `CPXgetbase` , are dependent on the solution type returned by `CPXsolninfo` .

The primal solution values of the variables are obtained by a call to `CPXgetx` , and then these values (along with the basis statuses if available) are printed, in a loop, for each variable. After that, a call to `CPXgetdblquality` provides a measure of the numerical roundoff error present in the solution, by obtaining the maximum amount by which any variable's lower or upper bound is violated.

After the `TERMINATE:` label, the data for the solution (`x` , `cstat` , and `rstat`) are freed. Then the problem object is freed by `CPXfreeprob`. After the problem is freed, the CPLEX environment is freed by `CPXcloseCPLEX`.

You can view the complete program online in the standard distribution of the product at *yourCPLEXinstallation*`/examples/src/lpex2.c`.

Adding rows to a problem: example lpex3.c

This application adds rows to a model.

This example illustrates how to develop your own solution algorithms with routines from the Callable Library. It also shows you how to add rows to a problem object. Here is the problem that `lpex3` solves:

Minimize	c^Tx
subject to	$Hx = d$
	$Ax = b$
	$l \leq x \leq u$

where	*H =*	*(-1 0 1 0 1 0 0 0)*	*d =*	*(-3)*
		(1 -1 0 1 0 0 0 0)		*(1)*
		(0 1 -1 0 0 1 -1 0)		*(4)*
		(0 0 0 -1 0 -1 0 1)		*(3)*
		(0 0 0 0 -1 0 1 -1)		*(-5)*
	A =	*(2 1 -2 -1 2 -1 -2 -3)*	*b =*	*(4)*
		(1 -3 2 3 -1 2 1 1)		*(-2)*
	c =	*(-9 1 4 2 -8 2 8 12)*		
	l =	*(0 0 0 0 0 0 0 0)*		
	u =	*(50 50 50 50 50 50 50 50)*		

The constraints Hx=d represent the flow conservation constraints of a pure network flow problem. The example solves this problem in two steps:

1. The CPLEX network optimizer is used to solve

Minimize	*c^Tx*
subject to	*Hx = d*
	l ≤ x ≤ u

2. The constraints Ax=b are added to the problem, and the dual simplex optimizer is used to solve the new problem, starting at the optimal basis of the simpler network problem.

The data for this problem consists of the network portion (using variable names beginning with the letter H) and the complicating constraints (using variable names beginning with the letter A).

The example first calls `CPXopenCPLEX` to create the environment and then turns on the CPLEX screen switch (`CPX_PARAM_SCRIND`). Next it sets the simplex display level (`CPX_PARAM_SIMDISPLAY`) to 2 to indicate iteration-by-iteration output, so that the progress of each iteration of the optimizer can be observed. Setting this parameter to 2 is not generally recommended; the example does so only for illustrative purposes.

The example creates a problem object by a call to `CPXcreateprob`. Then the network data is copied via a call to `CPXcopylp`. After the network data is copied, the parameter `CPX_PARAM_LPMETHOD` is set to `CPX_ALG_NET` and the routine `CPXlpopt` is called to solve the network part of the optimization problem using the network optimizer. The objective value of this problem is retrieved by `CPXgetobjval`.

Then the extra rows are added by `CPXaddrows`. For convenience, the total number of nonzeros in the rows being added is stored in an extra element of the array

rmatbeg , and this element is passed for the parameter nzcnt . The name arguments to CPXaddrows are NULL , since no variable or constraint names were defined for this problem.

After the CPXaddrows call, the parameter CPX_PARAM_LPMETHOD is set to CPX_ALG_DUAL and the routine CPXlpopt is called to re-optimize the problem using the dual simplex optimizer. After re-optimization, CPXsolution accesses the solution status, the objective value, and the primal solution. NULL is passed for the other solution values, since the information they provide is not needed in this example.

At the end, the problem is written as a SAV file by CPXwriteprob. This file can then be read into the Interactive Optimizer to analyze whether the problem was correctly generated. Using a SAV file is recommended over MPS and LP files, as SAV files preserve the full numeric precision of the problem.

After the TERMINATE: label, CPXfreeprob releases the problem object, and CPXcloseCPLEX releases the CPLEX environment.

You can view the complete program online in the standard distribution of the product at *yourCPLEXinstallation*/examples/src/lpex3.c.

Performing sensitivity analysis

This application demonstrates sensitivity analysis in the C API.

In “Performing sensitivity analysis” on page 33, there is a discussion of how to perform sensitivity analysis in the Interactive Optimizer. As with most interactive features of CPLEX, there is a direct approach to this task from the Callable Library. This section modifies the example lpex1.c in “Building and solving a small LP model in C” on page 90 to show how to perform sensitivity analysis with routines from the Callable Library.

To begin, make a copy of lpex1.c , and edit this new source file. Among the declarations (for example, immediately after the declaration for dj) insert these additional declarations:

```
double *lowerc = NULL, *upperc = NULL;
double *lowerr = NULL, *upperr = NULL;
```

At some point after the call to CPXlpopt, (for example, just before the call to CPXwriteprob), perform sensitivity analysis on the objective function and on the righthand side coefficients by inserting this fragment of code:

```
upperc = (double *) malloc (cur_numcols * sizeof(double));
lowerc = (double *) malloc (cur_numcols * sizeof(double));
status = CPXobjsa (env, lp, 0, cur_numcols-1, lowerc, upperc);
if ( status ) {
     fprintf (stderr, "Failed to obtain objective sensitivity.\n");
     goto TERMINATE;
}
printf ("\nObjective coefficient sensitivity:\n");
for (j = 0; j < cur_numcols; j++) {
  printf ("Column %d:  Lower = %10g  Upper = %10g\n",
          j, lowerc[j], upperc[j]);
}

upperr = (double *) malloc (cur_numrows * sizeof(double));
lowerr = (double *) malloc (cur_numrows * sizeof(double));
status = CPXrhssa (env, lp, 0, cur_numrows-1, lowerr, upperr);
if ( status ) {
  fprintf (stderr, "Failed to obtain RHS sensitivity.\n");
```

```
  goto TERMINATE;
}
printf ("\nRight-hand side coefficient sensitivity:\n");
for (i = 0; i < cur_numrows; i++) {
  printf ("Row %d:  Lower = %10g  Upper = %10g\n",
          i, lowerr[i], upperr[i]);
}
```

This sample is familiarly known as “throw away” code. For production purposes, you probably want to observe good programming practices such as freeing these allocated arrays at the `TERMINATE` label in the application.

A bound value of 1e+20 (`CPX_INFBOUND`) is treated as infinity within CPLEX, so this is the value printed by our sample code in cases where the upper or lower sensitivity range on a row or column is infinite; a more sophisticated program might print a string, such as `-inf` or `+inf`, when negative or positive `CPX_INFBOUND` is encountered as a value.

Similar code could be added to perform sensitivity analysis with respect to bounds via `CPXboundsa`.

Chapter 8. Python tutorial

Use CPLEX interactively in a Python session, or write an application using the Python API for CPLEX.

Design of CPLEX in a Python application

An application of CPLEX in the Python programming language uses Python objects.

The Python API of CPLEX consists of files in the directory where you installed the product (referred to here as `yourCPLEXHome`).

The primary class in the **module** cplex is the **class** Cplex. It encapsulates the mathematical formulation of an optimization problem together with information that you, the user, specify about how CPLEX should solve the problem. The class Cplex provides methods for modifying a problem, solving the problem, and querying both the problem itself and its solution. The methods of the class Cplex are grouped into categories of related functionality. For example, methods for adding, modifying, and querying data related to variables are contained in the member variables of the class Cplex.

Starting the CPLEX Python API

Use the Python environment variable `PYTHONPATH` to get started with the CPLEX Python API on your system.

In this tutorial, the directory where you find the CPLEX Python API (for example, as a feature of your installation of IBM ILOG CPLEX Optimization Studio) is known as `yourCPLEXhome`. In a path name such as `yourCPLEXhome/python/PLATFORM/`, the term `PLATFORM` represents the name of one of the various platforms on which the CPLEX Python API is available. In this context, a platform is a combination of operating system (such as a Microsoft Windows designation or a GNU/Linux distribution) and chip-type (such as 32- or 64-bit architecture) for which CPLEX is distributed as a feature. In that directory, you find these elements:

- a subdirectory named `cplex`
- a file named `setup.py`

Before you start this tutorial, you need to use that script to make your Python installation aware of your CPLEX installation. For more detail about using that script (and for an alternative to it), see “Setting up the Python API of CPLEX” on page 5.

Accessing the module `cplex`

Invoke CPLEX in a Python session.

After you have installed the CPLEX Python API on your system, you access CPLEX from the Python programming language through the cplex module. To do so, you may use any of the following commands either from a script written in the Python programming language or from within the Python interpreter:

```
import cplex
```

© Copyright IBM Corp. 1987, 2013

or

```
from cplex import *
```

or

```
import cplex as NAME
```

where NAME is any Python name of your choice.

Building and solving a small LP with Python

Solve a linear programming model using the CPLEX Python API.

After you have installed the CPLEX Python API on your system and opened a Python interactive session, you can actually build and solve a small LP model as a demonstration. The demonstration shows how to:

- model by rows;
- model by columns;
- model by nonzero elements.

The code sample, `lpex1.py`, is one of the examples in the standard distribution of the product. It is an extension of the example presented in *Getting Started with CPLEX* as “Problem statement” on page 11. The demonstration shows three different ways to define an LP problem through the CPLEX Python API, how to solve it, and how to access the solution. Here is the model that the sample optimizes:

```
Maximize
x1  + 2x2 + 3x3
subject to
–x1 +  x2 + x3 <= 20
 x1 – 3x2 + x3 <= 30
with these bounds
0 <= x1 <= 40
0 <= x2 <= infinity
0 <= x3 <= infinity
```

First, the example checks the correctness of command-line arguments. If the command-line arguments are valid, the example creates and solves the problem within a try/except clause to handle any errors that may occur. Then, depending on the command line argument, the example calls one of the functions `populatebyrow`, `populatebycolumn`, or `populatebynonzero`, to fill the Cplex object with a representation of the optimization problem. These functions generate lists containing the data that define the LP problem. After the Cplex object has been populated by one of these three functions, its method solve invokes the optimizer in this sample.

If the optimizer fails to generate a solution, CPLEX raises an exception.

If a solution is found, CPLEX prints solution information to `sys.stdout` when a user's application invokes print statements.

Reading and writing CPLEX models to files with Python

Read models from files and write models to files in an application using the CPLEX Python API.

In general, the methods of the class Cplex query, add, or modify data comprising an optimization problem.

Moreover, the class Cplex provides **output streams** for general log messages, results, warning messages, and error messages. A user sets those output streams by means of the methods `set_log_stream`, `set_results_stream`, `set_warning_stream`, and `set_error_stream`. These methods take either a file object or a filename as an argument; they also accept the argument `None` to disable their output.

These methods can also take, as an optional second argument, a **parsing function** that takes a string as input and returns a string as output. If the user specifies a parsing function, CPLEX uses that function to process the output to that stream.

By **default**, the log and results streams are set to `sys.stdout` and the error and warning streams are set to `sys.stderr`.

The sample `lpex1.py` contains one more line that deserves explanation in the context of reading and writing files:

```
my_prob.write("lpex1.lp")
```

That statement causes `my_prob` to write the data it has stored to the file named `lpex1.lp`. In this sample, the file is written in **LP format**. The reference manual, File formats supported by CPLEX, documents that file format with respect to CPLEX in the topic LP file format: algebraic representation.

Other formats supported for writing problems to a file are **MPS** and **SAV** (also documented by the reference manual, *File formats supported by CPLEX*, in the topic MPS file format: industry standard and in SAV.

The CPLEX Python API decides which file format to write based on the extension of the file name; you can also use an optional format string as an additional argument to specify a file format.

The class Cplex also supports reading of files through its method read and through its constructor. For example, if `cpx` is an instance of the class Cplex, then a call like this:

```
cpx.read("file.lp")
```

or like this:

```
cpx = cplex.Cplex("file.lp")
```

causes the CPLEX Python API to read a problem from the formatted file named `file.lp`.

Note:

The method `read` replaces all data currently stored in the object `cpx`.

MPS and SAV format files are also supported in a similar way.

Selecting an optimizer in Python

Select an optimizer using the CPLEX Python API, according to these criteria.

The CPLEX Python API provides a single method, solve, to optimize problems. That method uses the features of the model to deduce the appropriate algorithm for solving the problem. While the default optimizer works well for a wide variety of models, the CPLEX Python API allows you to control which algorithm to invoke for certain types of problems by your changing the values of certain parameters.

For example, if `cpx` is an instance of the class Cplex encapsulating a MIP (mixed integer programming) problem, you can specify which linear programming algorithm is used for solving the root problem and the node subproblems, like this:

```
cpx.parameters.mip.strategy.startalgorithm.set(start)
cpx.parameters.mip.strategy.subalgorithm.set(sub)
```

where `start` is one of the attributes of `cplex.parameters.mip.strategy.startalgorithm.values` for the root node and `sub` is one of the attributes of `cplex.parameters.mip.strategy.subalgorithm.values` for the subproblem nodes.

Example: reading a problem from a file lpex2.py

The sample lpex2.py illustrates reading a problem from a file.

The sample `lpex2.py` shows how to read an optimization problem from a file, and solve the problem with a specified optimizer option. If solution information or a simplex basis are available, the sample application prints them. Finally the sample prints the maximum infeasibility of any variable of the solution.

The user passes the name of the file to read and the choice of optimizers as arguments on the command line. For example, this command:

```
$ python lpex2.py example.mps d
```

reads the file `example.mps` and solves the problem with the dual simplex optimizer.

Alternatively, you may execute the statement:

```
import lpex2
```

within the Python interpreter and specify the problem file and optimizer interactively.

In summary, this example shows how to:

- read a model from a formatted file;
- select the optimizer;
- access basis information;
- query quality measures of the solution.

The general structure of this sample is similar to the sample `lpex1.py`. It starts by creating an instance of the class Cplex. A try/except statement encloses the code that follows to handle errors gracefully. You can see the complete application online in the standard distribution of the product at `yourCPLEXHome/examples/src/lpex2.py`.

Modifying and re-optimizing in the CPLEX Python API

Modify a model and re-optimize in the CPLEX Python API to see the effect.

In many situations, the solution to a model is only the first step. One of the important features of the CPLEX Python API is the ability to modify and then re-optimize the model even after it has been created and solved one or more times.

After CPLEX solves a problem, you can add, remove, or modify constraints to produce a different but related model. When you modify a problem, CPLEX tries to maintain as much information from the previous solution as reasonable and possible, in order to have a better start when it solves the modified problem. In particular, when solving LPs or QPs with a simplex optimmizer, CPLEX attempts to maintain a basis which will be used the next time the method solve is invoked, with the aim of making subsequent solves go faster.

Example: modifying a model lpex3.py

Modify the model in the sample lpex3.py.

The sample `lpex3.py` demonstrates how to:

- set CPLEX parameters;
- modify an optimization model;
- start optimization from an existing basis.

The problem that `lpex3.py` solves looks like this:

```
Minimize
c^Tx
subject to
Hx = d
Ax = b
l <= x <= u
where
H = ( -1 0 1 0 1 0 0 0 )
d =
( -3 )
( 1 -1 0 1 0 0 0 0 )
( 1 )
( 0 1 -1 0 0 1 -1 0 )
( 4 )
( 0 0 0 -1 0 -1 0 1 )
( 3 )
( 0 0 0 0 -1 0 1 -1 )
( -5 )
A = ( 2 1 -2 -1 2 -1 -2 -3 )
b =
( 4 )
( 1 -3 2 3 -1 2 1 1 )
( -2 )
c = (-9 1 4 2 -8 2 8 12 )
l = ( 0 0 0 0 0 0 0 0 )
u =(50 50 50 50 50 50 50 50 )
```

The constraints `Hx=d` represent the flow conservation of a pure network flow problem. The sample solves this problem in two steps:

- First, the CPLEX network optimizer solves:

  ```
  Minimize
  c^Tx
  subject to
  Hx = d
  l <= x <= u
  ```

- Second, the sample adds the constraints `Ax=b` to the model, and invokes the dual simplex optimizer to solve the full problem, starting from the optimal basis of the network problem.

The dual simplex optimizer is highly effective in such a case because this basis remains dual feasible after the slacks (artificial variables) of the added constraints are initialized as basic.

The 0 (zero) values in the data are omitted in the sample. CPLEX makes extensive use of sparse matrix methods and, although CPLEX correctly handles any explicit zero coefficients given to it, most programs solving models of more than modest size benefit (in terms of both storage space and speed) if the natural sparsity of the model is exploited from the very start.

Before solving the model, the sample selects the network optimizer by setting the parameter `lpmethod` to the value `parameters.lpmethod.values.network`.

The sample also sets the simplex display parameter so that the simplex optimizer logs information as it executes.

For an introduction to CPLEX parameters in the CPLEX Python API, see also the topic "Using CPLEX parameters in the CPLEX Python API."

Using CPLEX parameters in the CPLEX Python API

Manage CPLEX parameters in the CPLEX Python API.

For CPLEX users familiar with the Interactive Optimizer, setting and querying parameters in the CPLEX Python API is similar to parameter handling in the Interactive Optimizer. The class Cplex offers a data member named `parameters` containing names of parameters (such as `lpmethod` and `threads`) and names of groups of related parameters (such as `barrier` and `output`). These groups can in turn contain individual parameters and groups of parameters themselves. The hierarchy is the same as that found in the Interactive Optimizer, with the exception of parameters such as `output dialog`; their functionality is handled by other parts of the CPLEX Python API.

Tip:

If you are already familiar with the names of parameters in the Interactive Optimizer, then you quickly recognize names of parameters in the Python API. For example, the command "`set mip limits nodes 1`" in the Interactive Optimizer corresponds to "`c.parameters.mip.limits.nodes.set(1)`" in a Python session.

For users whose first contact with CPLEX is through the CPLEX Python API, the following topics introduce setting parameters, querying their current value, and using groups of parameters.

Setting and querying parameters in the CPLEX Python API

As objects themselves, parameters offer the following **methods**:

- `get()` returns the current value of the parameter.
- `set(value)` sets the invoking parameter to `value`. If `value` is of the wrong type for the invoking parameter, or if `value` is less than the minimum value or greater than the maximum value for the parameter, CPLEX raises an exception.

- `reset()` sets the parameter to its default value.
- `default()` returns the default value of the parameter.
- `type()` returns the type of the parameter.
- `help()` returns a brief description of the parameter.

Numerical parameters offer these additional methods:
- `min()` returns the minimum value allowed for the parameter.
- `max()` returns the maximum value allowed for the parameter.

Parameters of type `float` with no restrictions on their value return 0.0 (zero) when you invoke the method `min()` or `max()`.

Certain integer parameters, such as `lpmethod`, have values with a special meaning. Such parameters also have a **data attribute** named `values`, which has as its attributes the values that the parameter can take. For example, if `cpx` is an instance of the class Cplex encapsulating an LP problem, then:

```
>>> cpx.parameters.lpmethod.set(cpx.parameters.lpmethod.values.primal)
>>> cpx.solve()
```

results in CPLEX solving the problem by the primal simplex optimizer.

Parameter groups

For your convenience, CPLEX parameters are organized into groups of parameters that you can manage simultaneously. A **parameter group** is an instance of the class ParameterGroup.

The class `ParameterGroup` offers these methods:
- `reset` sets all parameters within the group to their default value.
- `get_changed` returns a list of pairs (parameter, current value) for the members of the group not currently at their default value.

The parameter group that encompasses all parameters is an instance of the class `RootParameterGroup`, a subclass of the class `ParameterGroup`. It offers these methods:
- `read(filename)` reads a set of parameters from the named file.
- `write(filename)` writes a set of parameters to the named file.
- `tune_problem(fixed_parameters_and_values=[])` tunes the parameters to improve performance on an instance of the class Cplex. The argument `fixed_parameters_and_values` is a sequence of pairs (parameter, value) as returned by the method `ParameterGroup.get_changed`.
- `tune_problem_set(filenames, filetypes=[], fixed_parameters_and_values=[])` tunes the parameters to improve performance on a set of problems. Again, the argument `fixed_parameters_and_values` is a sequence of pairs (parameter, value) as returned by the method `ParameterGroup.get_changed`.

Index

A

accessing
  basic rows and columns of solution in Interactive Optimizer 32
  basis information (C++ API) 61
  dual values in Interactive Optimizer 32
  dual values in Interactive Optimizer (example) 32
  module of Python API 97
  objective function value in Interactive Optimizer 32
  quality of solution in Interactive Optimizer 32
  reduced cost (Java API) 70
  reduced costs in Interactive Optimizer 32
  slack values in Interactive Optimizer 32
  solution values (C++ API) 54
  solution values in Interactive Optimizer 32
add Interactive Optimizer command 40, 41
  file name and 41
  syntax 41
add(obj) method (Java API) 68
adding
  bounds in Interactive Optimizer 40
  constraint to model (C++ API) 62
  constraints in Interactive Optimizer 40
  from a file in Interactive Optimizer 41
  interactively in Interactive Optimizer 41
  objective (shortcut) (Java API) 68
  objective function to model (C++ API) 52
  rows to a problem (C API) 92
addLe method (Java API) 72
addMinimize method (Java API) 68, 72
advanced basis
  advanced start indicator in Interactive Optimizer 31
algorithm
  creating object (C++ API) 54
  role in application (C++ API) 56
and method (Java API) 72
application
  and Callable Library vi
  and Concert Technology vi
  compiling and linking (C++ API) 50
  compiling and linking Callable Library (C API) 83
  compiling and linking Component Libraries 9
  development steps (C API) 87
  error handling (C API) 89
  error handling (C++ API) 55

B

baropt Interactive Optimizer command 32
barrier optimizer
  availability in Interactive Optimizer 32
BAS file format
  reading from Interactive Optimizer 38
  writing from Interactive Optimizer 36
basis
  accessing information (C++ API) 61
  basis information (Java API) 70
  starting from previous (C++ API) 63
basis file
  reading in Interactive Optimizer 38
  writing in Interactive Optimizer 36
Boolean parameter (C++ API) 63
Boolean variable
  representing in model (C++ API) 52
bound
  adding in Interactive Optimizer 40
  changing in Interactive Optimizer 43
  default values in Interactive Optimizer 23
  displaying in Interactive Optimizer 28
  entering in LP format in Interactive Optimizer 23
  removing in Interactive Optimizer 43
  sensitivity analysis in Interactive Optimizer 33
box variable in Interactive Optimizer 25

C

C++ API
  compiler options 2
  linker options 2
Callable Library
  description vi
  example model 14
Callable Library (C API) 83
  application development steps 87
  compiler options 2
  compiling and linking applications 83
  conceptual design 83
  CPLEX operation 85
  distribution file 84
  error handling 89
  linker options 2
  opening CPLEX 85
change Interactive Optimizer command 41
  bounds 43
  change options 42
  coefficient 44
  delete 44
change Interactive Optimizer command *(continued)*
    delete options 44
  objective 44
  rhs 44
  sense 42
  syntax 45
changing
  bounds in Interactive Optimizer 43
  coefficients in Interactive Optimizer 44
  constraint names in Interactive Optimizer 42
  objective in Interactive Optimizer 44
  parameters (C++ API) 63
  parameters in Interactive Optimizer 39
  problem in Interactive Optimizer 41
  righthand side (rhs) in Interactive Optimizer 44
  sense in Interactive Optimizer 42
  variable names in Interactive Optimizer 42
choosing
  optimizer (C API) 88
  optimizer (C++ API) 59
  optimizer in Interactive Optimizer 32
class library (Java API) 65
classpath (Java API) 66
  command line option 65
clean up data 45
coefficient
  changing in Interactive Optimizer 44
column
  expressions (C++ API) 57
command
  executing from operating system in Interactive Optimizer 46
  input formats in Interactive Optimizer 19
  Interactive Optimizer list 19
compiler
  -DNDEBUG option (C++ API) 55
  error messages (C++ API) 50
  Microsoft Visual C++ Command Line (C API) 85
  using with CPLEX (C++ API) 50
compiler option 2
compiling
  applications 9
  applications (C API) 83
  applications (C++ API) 50
Component Libraries
  defined vi
  running examples 8
  verifying installation 8
Concert Technology
  C++ classes 51
  C++ objects 49
  compiling and linking applications (C++ API) 50

© Copyright IBM Corp. 1987, 2013

Concert Technology *(continued)*
CPLEX design in (C++ API) 49
error handling (C++ API) 55
running examples (C++ API) 50
Concert Technology (C++ API) 49, 63
Concert Technology Library
description vi
example model 12
constraint
adding (C++ API) 62
adding in Interactive Optimizer 40
changing names in Interactive Optimizer 42
changing sense in Interactive Optimizer 42
creating (C++ API) 57
default names in Interactive Optimizer 23
deleting in Interactive Optimizer 44
displaying in Interactive Optimizer 28
displaying names in Interactive Optimizer 27
displaying nonzero coefficients in Interactive Optimizer 25
displaying number in Interactive Optimizer 25
displaying type in Interactive Optimizer 25
entering in LP format in Interactive Optimizer 22
name limitations in Interactive Optimizer 23
naming in Interactive Optimizer 23
range (C++ API) 57
constraints
adding to a model (Java API) 68
continuous variable
representing (C++ API) 52
CPLEX
compatible platforms vii
Component Libraries vi
description v
directory structure 6
problem types v
quitting in Interactive Optimizer 46
setting up 1
starting in Interactive Optimizer 19
technologies vi
cplex command in Interactive Optimizer 19
cplex.jar (location) 65
cplex.log file in Interactive Optimizer 31
CPXaddcols routine
example in C API 90
modular data in C API 88
populating problem (C API) 86
CPXaddrows routine
LP example in C API 90
modular data in C API 88
network example in C API 92
populating model in C API 86
CPXchgcoeflist routine
example in C API 90
modular data in C API 88
populating model in C API 86
CPXcloseCPLEX routine
LP example in C API 90
MPS example in C API 91
network example in C API 92
purpose in C API 85
CPXcopylp routine
building model in memory for C API 88
efficient arrays in C API 88
example in C API 92
not for changing model in C API 86
populating model in C API 86
CPXcreateprob routine
LP example in C API 91
network example in C API 92
purpose in C API 86
use in C API 86
CPXfreeprob routine
file format example in C API 91
LP example in C API 90
network example in C API 92
purpose in C API 86
CPXgeterrorstring routine
closing LP example in C API 90
opening LP example in C API 90
CPXgetobjval routine 92
CPXlpopt routine
LP example in C API 90
network example in C API 92
CPXmsg routine 85
CPXnewcols routine
LP example in C API 90
modular data in C API 88
populating model in C API 86
CPXnewrows routine 90
example in C API 90
modular data in C API 88
populating model in C API 86
CPXopenCPLEX routine
file format example in C API 91
LP example in C API 90
network example in C API 92
purpose in C API 85
CPXreadcopyprob routine
example in C API 91
formatted data files in C API 86
CPXsetintparam routine 90
CPXsolution routine
LP example in C API 90
network example in C API 92
CPXwriteprob routine
LP example in C API 90
network example in C API 92
testing in CAPI 89
creating
algorithm object (C++ API) 54, 56
automatic log file in Interactive Optimizer 31
binary problem representation (C API) 89
constraint (C++ API) 57
environment (C API) 92
environment object (C++ API) 51, 56
model (IloModel) (C++ API) 52
model (Java API) 68
model objects (C++ API) 56
objective function (C++ API) 57
creating *(continued)*
optimization model (C++ API) 52
problem files in Interactive Optimizer 34
problem object (C API) 86, 92

D

data
entering in Interactive Optimizer 24
entry options viii
deleting
constraints in Interactive Optimizer 44
problem options in Interactive Optimizer 44
variables in Interactive Optimizer 44
directory structure 6
display Interactive Optimizer command 24, 42
options 24
problem 24
bounds 28
constraints 28
names 27, 28
options 24
stats 25
syntax 25
sensitivity 33
syntax 34
settings 40
solution 32
syntax 33
specifying item ranges 26
syntax 29
displaying
basic rows and columns in Interactive Optimizer 32
bounds in Interactive Optimizer 28
constraint names in Interactive Optimizer 27
constraints in Interactive Optimizer 28
nonzero constraint coefficients in Interactive Optimizer 25
number of constraints in Interactive Optimizer 25
objective function in Interactive Optimizer 28
optimal solution in Interactive Optimizer 30
parameter settings in Interactive Optimizer 40
post-solution information in Interactive Optimizer 32
problem in Interactive Optimizer 24
problem options in Interactive Optimizer 24
problem part in Interactive Optimizer 25
problem statistics in Interactive Optimizer 25
sensitivity analysis (C API) 94
sensitivity analysis in Interactive Optimizer 33
slack values in Interactive Optimizer 32

displaying *(continued)*
  solution values in Interactive Optimizer 32
  type of constraint in Interactive Optimizer 25
  variable names in Interactive Optimizer 27
  variables in Interactive Optimizer 25
dual simplex optimizer
  as default in Interactive Optimizer 30
  availability in Interactive Optimizer 32
  finding a solution (C API) 90
dual values
  accessing (Java API) 70
  accessing in Interactive Optimizer 32

E

Eclipse
  configuring for CPLEX applications 4
enter Interactive Optimizer command 21
  bounds 23
  maximize 22
  minimize 22
  subject to 22, 41
  syntax 22
entering
  bounds in Interactive Optimizer 23
  constraint names in Interactive Optimizer 23
  constraints in Interactive Optimizer 22
  example problem in Interactive Optimizer 21
  item ranges in Interactive Optimizer 26
  keyboard data in Interactive Optimizer 24
  objective function in Interactive Optimizer 22, 23
  objective function names in Interactive Optimizer 23
  problem in Interactive Optimizer 21, 22
  problem name in Interactive Optimizer 21
  variable bounds in Interactive Optimizer 23
  variable names in Interactive Optimizer 22
environment object
  creating (C++ API) 51, 56
  destroying (C++ API) 51
  memory management and (C++ API) 51
equality constraints
  adding to a model (Java API) 68
error
  invalid encrypted key (Java API) 66
  NoClassDefFoundError (Java API) 66
  UnsatisfiedLinkError (Java API) 66
error handling
  compiler (C++ API) 50
  linker (C++ API) 50
  programming errors (C++ API) 55
  runtime errors (C++ API) 55
  testing installation 8
  testing installation (C++ API) 50
example
  adding rows to a problem (C API) 92
  entering a problem in Interactive Optimizer 21
  entering and optimizing a problem (C API) 90
  entering and optimizing a problem in C# 78
  ilolpex2.cpp (C++ API) 60
  ilolpex3.cpp (C++ API) 62
  lpex1.c (C API) 90
  lpex1.cs 78
  lpex2.c (C API) 91
  lpex2.py 100
  lpex3.c (C API) 92
  lpex3.py 101
  modifying an optimization problem (C++ API) 62
  reading a problem file (C API) 91
  reading a problem from a file (C++ API) 60
  running (C++ API) 50
  running Callable Library (C API) 84
  running Component Libraries 8
  running from standard distribution (C API) 84
  solving a problem in Interactive Optimizer 30
exception handling (C++ API) 55
executing operating system commands in Interactive Optimizer 46
exportModel method
  IloCplex class (C++ API) 58
expression
  column 57

F

false return value of IloCplex.solve (Java API) 69
feasible solution (Java API) 69
file format
  read options in Interactive Optimizer 36
  write options in Interactive Optimizer 34
file name
  extension 58
  extension in Interactive Optimizer 35, 38

G

getCplexStatus method
  IloCplex class (C++ API) 54
getCplexStatus method (Java API) 69
getDuals method
  IloCplex class (C++ API) 56
getObjValue method
  IloCplex class (C++ API) 54
getReducedCosts method
  IloCplex class (C++ API) 56
getSlacks method
  IloCplex class (C++ API) 56
getStatus method
  IloCplex class (C++ API) 54, 56
getStatus method (Java API) 69
getValue method
  IloCplex class (C++ API) 54
getValues method
  IloCplex class (C++ API) 56
greater than equal to constraints
  adding to a model (Java API) 68

H

handle class
  definition (C++ API) 51
  empty handle (C++ API) 52
handling
  errors (C API) 89
  errors (C++ API) 55
  exceptions (C++ API) 55
help Interactive Optimizer command 19
histogram in Interactive Optimizer 29

I

IloAddNumVar class (C++ API) 57
IloColumn.and method (Java API) 72
IloCplex class (C++ API) 49
  exportModel method 58
  getCplexStatus method 54
  getDuals method 56
  getObjValue method 54
  getReducedCosts method 56
  getSlacks method 56
  getStatus method 54, 56
  getValue method 54
  getValues method 56
  importModel method 58, 60
  setParam method 59
  setRootAlgorithm method 61
  solve method 54, 56, 61
  solving with 54
IloCplex class (Java API) 65
  add modeling object 68
  addLe method 72
  addMinimize method 72
  numVarArray method 72
  prod method 72
  scalProd method 72
  sum method 72
IloEnv class (C++ API) 51
  end method 51
IloException class (C++ API) 55
IloExpr class (C++ API) 52
IloExtractable class (C++ API) 52
IloLinearNumExpr class (Java API) 68
IloMinimize function (C++ API) 52
IloModel class (C++ API)
  add method 52
  extractable 52
  role 49
IloModel class (Java API)
  column method 72

IloModel class (Java API) *(continued)*
numVar method 72
IloNumArray class (C++ API) 56
IloNumColumn class (C++ API) 57
IloNumExpr class (Java API) 68
IloNumVar class (C++ API) 57
columns and 57
IloObjective class (C++ API) 57
setLinearCoef method 58
IloRange class (C++ API)
casting operator for 57
example 52
setLinearCoef method 58
IloRange class (Java API)
setExpr method 73
importModel method
IloCplex class (C++ API) 58, 60
infeasible (Java API) 69
installing
Python API 97
installing CPLEX
testing installation 8
integer parameter (C++ API) 63
integer variable
optimizer used (C API) 88
representing in model (C++ API) 52
Interactive Optimizer 19, 46
command formats 19
commands 19
description vi
example model 11
quitting 46
starting 19
starting (tutorial) 19
invalid encrypted key (Java API) 66
iteration log in Interactive Optimizer 30, 31

J

Java API
setting up Eclipse 4
Java Native Interface (JNI) 65
Java Virtual Machine (JVM) 65
javamake for Windows 66

L

libformat (Java API) 66
linear optimization v
linker
error messages (C++ API) 50
using with CPLEX (C++ API) 50
linker option 2
linking
applications 9
applications (C++ API) 50
Callable Library (C API) applications 83
log file
adding to in Interactive Optimizer 39
cplex.log in Interactive Optimizer 31
creating in Interactive Optimizer 31
iteration log in Interactive Optimizer 30, 31
LP
creating a model 11
node (C++ API) 59
problem format v
root (C++ API) 59
solving a model 11
solving pure (C++ API) 59
LP file
format in Interactive Optimizer 22
reading in Interactive Optimizer 37
writing in Interactive Optimizer 35
lpex1.c
example (C API) 90
LPex1.java example 70
lpex2.py example 100
lpex3.py example 101
LPMETHOD parameter in Interactive Optimizer 30

M

make file target 2
makefile (Java API) 66
maximization in LP problem in Interactive Optimizer 22
memory management
by environment object (C++ API) 51
minimization in LP problem in Interactive Optimizer 22
MIP
description v
optimizer in Interactive Optimizer 32
solving (C++ API) 59
mipopt Interactive Optimizer command 32
model
adding constraints (C++ API) 62
creating (C++ API) 52
creating IloModel (C++ API) 52
creating objects in (C++ API) 56
extracting (C++ API) 56
modifying (C++ API) 61
reading from file (C++ API) 58, 60
solving (C++ API) 61
writing to file (C++ API) 58
modeling
by columns (C++ API) 57
by columns (Java API) 72
by nonzeros (C++ API) 58
by nonzeros (Java API) 73
by rows (Java API) 72
objects (C++ API) 49
variables (Java API) 68
modeling by rows (C++ API) 57
modifying
problem object (C API) 86
monitoring iteration log in Interactive Optimizer 30
MPS file format in Interactive Optimizer 38

N

netopt Interactive Optimizer command 32
network
description v
network flow (C++ API) 62
network optimizer
availability in Interactive Optimizer 32
solving with (C++ API) 62
Nmake (Java API) 66
NoClassDefFoundError (Java API) 66
node LP
solving (C++ API) 59
nonzeros
modeling (C++ API) 58
modeling (Java API) 73
notation in this manual ix
notification (C++ API) 61
numeric parameter (C++ API) 63
numVarArray method (Java API) 72

O

objective function
accessing value in Interactive Optimizer 32
adding to model (C++ API) 52
changing coefficient in Interactive Optimizer 44
changing sense in Interactive Optimizer 42
creating (C++ API) 57
default name in Interactive Optimizer 23
displaying in Interactive Optimizer 28
entering in Interactive Optimizer 23
entering in LP format in Interactive Optimizer 22
name in Interactive Optimizer 23
sensitivity analysis in Interactive Optimizer 33
operator() (C++ API) 57
operator+ (C++ API) 57
optimal solution (Java API) 69
optimization model
creating (C++ API) 52
defining extractable objects (C++ API) 52
extracting (C++ API) 52
optimization problem
interrupting in Interactive Optimizer 32
reading from file (C++ API) 60
representing (C++ API) 56
solving with IloCplex (C++ API) 54
optimize Interactive Optimizer command 30
re-solving 31
syntax 31
optimizer
choosing (Python API) 100
choosing by problem type (C API) 88
choosing by switch in application (C++ API) 61
choosing in Interactive Optimizer 32
options vii
syntax for choosing (C++ API) 59

ordering variables in Interactive Optimizer 28
output method (Java API) 70
OutputStream (Java API) 70

P

parallel
- choosing optimizers for viii

parameter
- Boolean (C++ API) 63
- changing (C++ API) 63
- changing in Interactive Optimizer 39
- displaying settings in Interactive Optimizer 40
- in Python API 102
- integer (C++ API) 63
- list of settable in Interactive Optimizer 39
- numeric (C++ API) 63
- resetting to defaults in Interactive Optimizer 40
- string (C++ API) 63

parameter group (Python API) 103
parameter specification file in Interactive Optimizer 40
path names in Interactive Optimizer 36
populateByColumn method (Java API) 70
populateByNonzero method (Java API) 73
populateByNonzero method(Java API) 70
populateByRow (Java API) 70
primal simplex optimizer
- availability in Interactive Optimizer 32

primopt Interactive Optimizer command 32
problem
- change options in Interactive Optimizer 42
- changing in Interactive Optimizer 41
- creating binary representation (C API) 89
- data entry options viii
- displaying a part in Interactive Optimizer 25
- displaying in Interactive Optimizer 24
- displaying options in Interactive Optimizer 24
- displaying statistics in Interactive Optimizer 25
- entering from the keyboard in Interactive Optimizer 21
- entering in LP format in Interactive Optimizer 22
- naming in Interactive Optimizer 21
- reading files (C API) 91
- solving (C API) 90
- solving in Interactive Optimizer 30
- verifying entry in Interactive Optimizer 24, 42

problem file
- reading in Interactive Optimizer 36
- writing in Interactive Optimizer 34

problem formulation
- ilolpex1.cpp (C++ API) 56
- Interactive Optimizer and 21
- lpex1.c (C API) 90
- lpex1.cs 78
- LPex1.java (Java API) 70
- standard notation for v

problem object
- creating (C API) 86
- modifying (C API) 86

problem types solved by CPLEX v
Python
- tutorial 97

Python API 97
- accessing cplex module 97
- modifying model 101
- paramters in 102
- re-optimizing problem 101
- reading files in 99
- selecting optimizer 100
- setting up on GNU/Linux 5
- setting up on UNIX 5
- setting up on Windows 5
- writing files from 99

Q

QCP
- description v
- optimizer for vii

QP
- applicable algorithms (C++ API) 59
- description v
- solving pure (C++ API) 59

querying
- parameters (Python API) 102

quit Interactive Optimizer command 46
quitting
- CPLEX in Interactive Optimizer 46
- Interactive Optimizer 46

R

range constraint
- adding to a model (Java API) 68

range constraint (C++ API) 57
re-solving in Interactive Optimizer 31
read Interactive Optimizer command 36, 37, 38
- basis files and 38
- file type options 36
- syntax 39

reading
- file format in Interactive Optimizer 36
- LP files in Interactive Optimizer 37
- model from file (C++ API) 58, 60
- MPS files in Interactive Optimizer 38
- problem files (C API) 91
- problem files in Interactive Optimizer 36

reduced cost
- accessing (Java API) 70
- accessing in Interactive Optimizer 32

removing bounds in Interactive Optimizer 43
representing optimization problem (C++ API) 56
righthand side (RHS)
- changing coefficient in Interactive Optimizer 44
- sensitivity analysis in Interactive Optimizer 33

root LP
- solving (C++ API) 59

S

SAV file format (C API) 92
saving
- problem files in Interactive Optimizer 34
- solution files in Interactive Optimizer 34

scalProd method (Java API) 72
sense
- changing in Interactive Optimizer 42

sensitivity analysis
- performing (C API) 94
- performing in Interactive Optimizer 33

set Interactive Optimizer command 39
- advance 31
- available parameters 39
- defaults 40
- logfile 31
- simplex 30
- syntax 40

setOut method (Java API) 70
setRootAlgorithm method
- IloCplex class (C++ API) 61

setting
- parameters (C++ API) 63
- parameters (Python API) 102
- parameters in Interactive Optimizer 39
- parameters to default in Interactive Optimizer 40

setting up CPLEX 1
- GNU/Linux 2

setWarning method (Java API) 70
shadow price
- accessing in Interactive Optimizer 32

slack
- accessing (Java API) 70
- accessing in Interactive Optimizer 32
- accessing values in Interactive Optimizer 32

SOCP
- description v
- optimizer for vii

solution
- accessing basic rows and columns in Interactive Optimizer 32
- accessing values (C++ API) 54
- accessing values in Interactive Optimizer 32
- displaying basic rows and columns in Interactive Optimizer 32
- displaying in Interactive Optimizer 32
- outputting (C++ API) 56
- process in Interactive Optimizer 30

solution (continued)
querying results (C++ API) 54
reporting optimal in Interactive Optimizer 30
restarting in Interactive Optimizer 31
sensitivity analysis (C API) 94
sensitivity analysis in Interactive Optimizer 33
solution file
writing in Interactive Optimizer 34
solve method
IloCplex class (C++ API) 54, 56, 61
solve method (Java API) 69, 70
solving
model (C++ API) 54, 61
node LP (C++ API) 59
problem (C API) 90
problem in Interactive Optimizer 30
root LP (C++ API) 59
with network optimizer (C++ API) 62
sparse matrix (C++ API) 62
starting
CPLEX in Interactive Optimizer 19
from previous basis (C++ API) 63
Interactive Optimizer 19
new problem in Interactive Optimizer 21
string parameter (C++ API) 63
structure of a CPLEX application (Java API) 67
Supported Platforms (Java API) 66
System.out method (Java API) 70

T

tranopt Interactive Optimizer command 32

U

unbounded (Java API) 69
UNIX
building Callable Library (C API) applications 84
executing commands in Interactive Optimizer 46
testing CPLEX (C++ API) 50
verifying set-up 8
UnsatisfiedLinkError (Java API) 66

V

variable
Boolean (C++ API) 52
box in Interactive Optimizer 25
changing bounds in Interactive Optimizer 43
changing names in Interactive Optimizer 42
continuous (C++ API) 52
deleting in Interactive Optimizer 44
displaying in Interactive Optimizer 25
displaying names in Interactive Optimizer 27
variable (continued)
entering bounds in Interactive Optimizer 23
entering names in Interactive Optimizer 22
integer (C++ API) 52
modeling (Java API) 68
name limitations in Interactive Optimizer 22
ordering in Interactive Optimizer 28
removing bounds in Interactive Optimizer 43

W

warning method (Java API) 70
wildcard
displaying ranges of items in Interactive Optimizer 26
solution information in Interactive Optimizer 33
wildcard in Interactive Optimizer 26
Windows
building Callable Library (C API) applications 84
dynamic loading (C API) 85
Microsoft Visual C++ compiler (C API) 85
Microsoft Visual C++ IDE (C API) 84
setting up CPLEX 1
testing CPLEX (C++ API) 50
verifying set-up 8
write Interactive Optimizer command 34, 35
file type options 34
syntax 36
writing
basis files in Interactive Optimizer 36
file format for Interactive Optimizer 34
LP files in Interactive Optimizer 35
model to file (C++ API) 58
problem files in Interactive Optimizer 34
solution files in Interactive Optimizer 34

X

xecute Interactive Optimizer command 46
syntax 46

Printed in USA