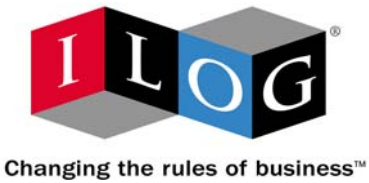

ILOG CPLEX 9.0

User's Manual

October 2003

Copyright © 1987-2003, ILOG, S. A. — All rights reserved.

Table of Contents

List of Figures **19**
List of Tables **21**
List of Code Samples **23**

Preface **Meet ILOG CPLEX** **25**

What Is ILOG CPLEX? **26**
What Does ILOG CPLEX Do? **26**
What You Need to Know **28**
In This Manual **28**
Examples Online 31
Notation in This Manual 33
Related Documentation **34**
Announcements: cplex-list@ilog.fr **36**
Further Reading **36**

Part I **Languages and APIs** **39**

Chapter 1 **ILOG Concert Technology for C++ Users** **41**

The Design of ILOG CPLEX Concert Technology **41**
Licenses 42
Compiling and Linking 42
Creating an Application with Concert Technology **43**

Modeling an Optimization Problem with Concert Technology 43
Modeling Classes 44
Data Management Classes 47
Solving Concert Technology Models with IloCplex 47
Extracting a Model 49
Solving a Model 50
Choosing an Optimizer 51
Controlling the Optimizers 52
Accessing Solution Information 54
Accessing Solution Status 54
Querying Solution Data 55
Accessing Basis Information 56
Performing Sensitivity Analysis 56
Analyzing Infeasible Problems 57
Solution Quality 57
Modifying a Model 58
Deleting and Removing Modeling Objects 58
Changing Variable Type 59
Handling Errors 60
Example: Optimizing the Diet Problem in C++ 61
Problem Representation 62
Application Description 64
Solving the Model with IloCplex 66
Complete Program 67

Chapter 2 ILOG Concert Technology for Java Users 73

Architecture of a CPLEX Java Application 74
Modeling with Concert Technology 75
Modeling with IloModeler 76
The Active Model 79
Building the Model 79
Solving the Model 81

Accessing Solution Information 82
Choosing an Optimizer 83
Solving a Single Continous Model 85
Solving Subsequent Continuous Relaxations in a MIP 85
Controlling ILOG CPLEX Optimizers 86
Parameters 86
Priority Orders and Branching Directions 87
More Solution Information 88
Writing Solution Files 88
Dual Solution Information 89
Basis Information 89
Infeasible Solution Information 89
Solution Quality 90
Advanced Modeling with IloLPMatrix 91
Modeling by Column 92
Optimizing the Diet Problem in Java 93
Creating the Optimizer 93
Example: Diet.java Source Code 94
Modifying the Model 98

Chapter 3 ILOG Concert Technology for C#.NET Users 101

Describe 102
Model 104
Build by Rows 105
Build by Columns 107
Solve 108
Good Programming Practices 109
Example: Optimizing the Diet Problem in C#.NET 111

Chapter 4 The ILOG CPLEX Callable Library 113

Architecture of the ILOG CPLEX Callable Library 113
Licenses 114

Compiling and Linking 115
Using the Callable Library in an Application 115
Initialize the ILOG CPLEX Environment 115
Instantiate the Problem Object 116
Put Data in the Problem Object 116
Optimize the Problem 117
Change the Problem Object 117
Destroy the Problem Object 117
Release the ILOG CPLEX Environment 117
ILOG CPLEX Programming Practices 118
Variable Names and Calling Conventions 118
Data Types 119
Ownership of Problem Data 120
Problem Size and Memory Allocation Issues 120
Status and Return Values 121
Symbolic Constants 121
Parameter Routines 121
Null Arguments 122
Row and Column References 122
Character Strings 123
Checking Problem Data 123
Callbacks 124
Portability 125
FORTRAN Interface 126
C++ Interface 127
Managing Parameters from the Callable Library 127
Example: Optimizing the Diet Problem in the Callable Library 129
Problem Representation 129
Program Description 131
Complete Program 132

Part II Programming Considerations 143

Chapter 5 Developing CPLEX Applications 145

Tips for Successful Application Development 145

Prototype the Model 145

Identify Routines to Use 146

Test Interactively 146

Assemble Data Efficiently 146

Test Data 147

Choose an Optimizer 147

Program with a View toward Maintenance and Modifications 148

Using the Interactive Optimizer for Debugging 150

Eliminating Common Programming Errors 151

Check Your Include Files 152

Clean House and Try Again 152

Read Your Messages 152

Check Return Values 152

Beware of Numbering Conventions 153

Make Local Variables Temporarily Global 153

Solve the Problem You Intended 153

Special Considerations for Fortran 153

Tell Us 153

Chapter 6 Managing Input and Output 155

Understanding File Formats 156

Working with LP Files 156

Working with MPS Files 157

Converting File Formats 158

Using Concert XML Extensions 159

Using Concert csvReader 160

Managing Log Files 161

Creating, Renaming, Relocating Log Files 161

Closing Log Files 161
Controlling Message Channels. 162
Parameter for Output Channels 162
Callable Library Routines for Message Channels 163
Example: Callable Library Message Channels 164

Chapter 7 Licensing an Application 175

Types of ILM Runtime Licenses 176
File-Based RTNODE, RTSTOKEN or TOKEN Keys. 176
Memory-Based RUNTIME Keys. 176
Routines and Methods Used for Licensing 176
Examples 177
CPXputenv Routine 177
CPXRegisterLicense Routine. 178
The registerLicense Method for Java Users 179
Summary. 179

Part III Continuous Optimization. 181

Chapter 8 Solving LPs: Simplex Optimizers 183

Choosing an Optimizer for Your LP Problem 184
Automatic Selection of Optimizer 185
Dual Simplex Optimizer 185
Primal Simplex Optimizer 186
Network Optimizer 186
Barrier Optimizer 186
Sifting Optimizer. 186
Concurrent Optimizer 187
Parameter Settings and Optimizer Choice. 187
Tuning LP Performance. 187
Preprocessing 188
Starting from an Advanced Basis. 190

Simplex Parameters 191
Diagnosing Performance Problems 195
Lack of Memory 195
Numerical Difficulties 196
Diagnosing LP Infeasibility 200
The Effect of Preprocessing on Feasibility 201
Coping with an Ill-Conditioned Problem or Handling Unscaled Infeasibilities 202
Interpreting Solution Statistics 203
Finding a Set of Irreducibly Inconsistent Constraints 204
More about Infeasibility: FeasOpt 210
Example: Using a Starting Basis in an LP Problem 215
Example ilolpex6.cpp 216
Example lpex6.c 218

Chapter 9 Solving LPs: Barrier Optimizer 225

Identifying LPs for Barrier Optimization 226
Using the Barrier Optimizer 228
Special Options 229
Controlling Crossover 229
Using VEC File Format 230
Interpreting the Barrier Log File 230
Preprocessing in the Log File 231
Nonzeros in Lower Triangle of AA^T in the Log File 231
Ordering-Algorithm Time in the Log File 231
Cholesky Factor in the Log File 232
Iteration Progress in the Log File 232
Infeasibility Ratio in the Log File 233
Understanding Solution Quality from the Barrier LP Optimizer 233
Tuning Barrier Optimizer Performance 235
Out-of-Core Barrier: Letting the Optimizer Use Disk for Storage 236
Preprocessing 236
Detecting and Eliminating Dense Columns 237

Choosing an Ordering Algorithm 238
Using a Starting-Point Heuristic 238
Overcoming Numerical Difficulties 239
Difficulties in the Quality of Solution 239
Difficulties during Optimization 241
Difficulties with Unbounded Problems 242
Diagnosing Barrier Optimizer Infeasibility 242

Chapter 10 Solving Network-Flow Problems 245

Choosing an Optimizer: Network Considerations 245
Formulating a Network Problem 246
Example: Network Optimizer in the Interactive Optimizer 247
Understanding the Network Log File 248
Tuning Performance of the Network Optimizer 249
Solving Problems with the Network Optimizer 249
Network Extraction 250
Preprocessing and the Network Optimizer 251
Example: Using the Network Optimizer with the Callable Library netex1.c 251
Solving Network-Flow Problems as LP Problems 258
Example: Network to LP Transformation netex2.c 259

Chapter 11 Solving Problems with a Quadratic Objective (QP) 265

Identifying Convex QPs 266
Entering QPs 267
Saving QP Problems 268
Changing Problem Type in QPs 269
Changing Quadratic Terms 270
Optimizing QPs 271
Diagnosing QP Infeasibility 272
Example: Creating a QP, Optimizing, Finding a Solution 273
Example: iloqpex1.cpp 274
Example: QPex1.java 276

Example: qpex1.c. 278
Example: Reading a QP from a File qpex2.c 286

Chapter 12 Solving Problems with Quadratic Constraints (QCP) 295

Identifying a Quadratically Constrained Program (QCP) 296
Convexity 296
Semi-definiteness. 298
Determining Problem Type 299
Changing Problem Type 299
Changing Quadratic Constraints 300
Solving with Quadratic Constraints 300
Numerical Difficulties and Quadratic Constraints 301
Examples: QCP 301

Part IV Discrete Optimization 303

Chapter 13 Solving Mixed Integer Programming Problems (MIP) 305

Stating a MIP Problem 306
Considering Preliminary Issues 307
Entering MIP Problems 307
Displaying MIP Problems 308
Changing Problem Type in MIPs 309
Using the Mixed Integer Optimizer 311
Feasibility and Optimality 311
Performance Features of the Mixed Integer Optimizer 313
Cuts 318
Parameters Affecting Cuts 320
Heuristics 321
Probing. 322
Preprocessing: Presolver and Aggregator 322
Starting from a Solution 324
Priority Orders 324

Termination 325
Using the MIP Solution 327
Using Special Ordered Sets 329
Example: SOS Type 1 for Sizing a Warehouse 329
Declaring SOS Members 330
Setting Branching Priority for an SOS 330
Assigning SOS Weights 331
Using Semi-Continuous Variables 331
Progress Reports: Interpreting the Node Log 331
Troubleshooting MIP Performance Problems 335
Too Much Time at Node 0 336
Trouble Finding More than One Feasible Solution 337
Large Number of Unhelpful Cuts 337
Lack of Movement in the Best Node 337
Time Wasted on Overly Tight Optimality Criteria 338
Slightly Infeasible Integer Variables 339
Running Out of Memory 340
Difficulty Solving Subproblems 343
Subproblem Optimization 344
Example: Optimizing a Basic MIP Problem 346
Complete Program: ilomipex1.cpp 346
Complete Program: mipex1.c 347
Example: Reading a MIP Problem from a File 356
Complete Program: ilomipex2.cpp 356
Complete Program: mipex2.c 357
Example: Using SOS and Priority 362
Complete Program: ilomipex3.cpp 362
Complete Program: mipex3.c 364

Chapter 14 Cutting Stock: Column Generation 375

What Is Column Generation? 375
Column-Wise Models in Concert Technology 376

Describing the Problem 376
Representing the Data 378
Developing the Model: Building and Modifying 378
Adding Extractable Objects: Both Ways 379
Adding Columns to a Model 380
Changing Coefficients of an Objective Function 380
Changing the Type of a Variable 381
Cut Optimization Model 381
Pattern Generator Model 381
Solving the Problem: Using More than One Algorithm 382
Ending the Program 383
Complete Program 384

Chapter 15 Rates: Using Semi-Continuous Variables 389

What Is IloCplex? 389
IloCplex and Extraction 389
IloCplex and MP Models 390
IloCplex and Algorithms 390
What Are Semi-Continuous Variables? 391
Describing the Problem 392
Representing the Problem 392
Building a Model 392
Solving the Problem 393
Ending the Program 393
Complete Program 393

Chapter 16 Transport: Piecewise Linear Optimization 397

Piecewise Linearity in ILOG CPLEX 397
What Is a Piecewise Linear Function? 397
Syntax of Piecewise Linear Functions 398
Discontinuous Piecewise Linear Functions 399
Using IloPiecewiseLinear 400

Special Considerations about Solving with IloPiecewiseLinear 400
Describing the Problem 401
Variable Shipping Costs 402
Model with Varying Costs 404
Representing the Data 405
Developing a Model 405
Adding Constraints 405
Checking Convexity and Concavity 406
Adding an Objective 406
Solving the Problem 407
Displaying a Solution 407
Ending the Application 407
Complete Program: transport.cpp 408

Chapter 17 Logical Constraints in Optimization 411

What Are Logical Constraints? 412
What Can CPLEX Extract from a Model with Logical Constraints? 412
Which Logical Expressions Can CPLEX Extract? 413
What Does CPLEX Transform? 414
Are Logical Constraints Discrete or Continuous? 416

Chapter 18 Food Manufacturing 2: Using Logical Constraints 419

Describing the Problem 419
Representing the Data 420
What Is Known? 420
What Is Unknown? 421
What Are the Constraints? 421
What Is the Objective? 423
Developing the Model 423
Using Logical Constraints 424
Solving the Problem 424
Ending the Program 425

Example: foodmanufact.cpp 425

Chapter 19 Early Tardy Scheduling 429

Describing the Problem 430
Understanding the Data File 430
Reading the Data 431
Creating Variables 431
Stating Precedence Constraints 432
Stating Resource Constraints 432
Representing the Piecewise Linear Cost Function 433
Transforming the Problem 433
Solving the Problem 434
Example: etsp.cpp 434

Part V Advanced Programming Techniques 439

Chapter 20 Using Query Routines in the Callable Library 441

Using Surplus Arguments for Array Allocations 441
Example: Using Query Routines lpex7.c 443

Chapter 21 Using Goals 453

Branch & Cut in ILOG CPLEX 454
The Goal Stack 460
Memory Management and Goals 461
Cuts and Goals 463
Injecting Heuristic Solutions 468
Controlling Goal-Defined Search 469
Search Limits 475

Chapter 22 Using Callbacks 477

Diagnostic Callbacks 478
Implementing Callbacks in ILOG CPLEX with Concert Technology 478
Writing Callback Classes by Hand 479

Writing Callbacks with Macros 480
Callback Interface 482
The Continuous Callback 482
Example: Deriving the Simplex Callback ilolpex4.cpp 482
Implementing Callbacks in the Callable Library 486
Setting Callbacks 486
Callbacks for LP/QP/QCPs and for MIPs 486
Return Values for Callbacks 487
Interaction Between Callbacks and ILOG CPLEX Parallel Optimizers 487
Example: Using Callbacks lpex4.c 488
Control Callbacks for IloCplex 496
Example: Controlling Cuts iloadmipex5.cpp 497

Chapter 23 Goals and Callbacks: a Comparison 505

Chapter 24 User Cut and Lazy Constraint Pools 507

What Are Pools of User Cuts or Lazy Constraints? 507
Adding User Cuts and Lazy Constraints 509
Component Libraries 509
Reading LP and SAV Files 509
Writing LP and SAV Files 510
Using the Interactive Optimizer 510
Deleting User Cuts and Lazy Constraints 511

Chapter 25 Advanced MIP Control Interface 513

Introduction to MIP Callbacks 514
Heuristic Callback 515
Cut Callback 516
Branch Selection Callback 517
Incumbent Callback 518
Node Selection Callback 519
Solve Callback 519

Chapter 26 Advanced Presolve Routines 521

Introduction to Presolve 521
Restricting Presolve Reductions 523
Manual Control of Presolve. 526
Modifying a Problem 528

Chapter 27 Parallel Optimizers 531

Threads 532
Example: Threads and Licensing 533
Threads and Performance Considerations 533
Nondeterminism. 533
Clock Settings and Time Measurement 534
Using Parallel Optimizers in the Interactive Optimizer. 534
Using Parallel Optimizers in the ILOG CPLEX Component Libraries 535
Parallel MIP Optimizer 535
Memory Considerations and the Parallel MIP Optimizer 537
Output from the Parallel MIP Optimizer 538
Parallel Barrier Optimizer 540
Concurrent Optimizer 540

Index 541

List of Figures

1.1 A View of Concert Technology for C++ Users **42**
2.1 A View of Concert Technology for Java Users **74**
4.1 A View of the ILOG CPLEX Callable Library **114**
6.1 ILOG CPLEX Message Handling Routines **164**
10.1 A Directed Network with Arc-Capacity, Flow-Cost, Sinks, and Sources **248**
11.1 Minimize a Convex Objective Function, Maximize a Concave Objective Function **266**
12.1 $x^2 + y^2 \leq 1$ is convex **296**
12.2 $x^2 + y^2 \geq 1$ is not convex **297**
12.3 $x^2 + y^2 = 1$ is not convex **298**
14.1 Two different patterns from a roll of stock **377**
16.1 A piecewise linear function with breakpoints **398**
16.2 A discontinuous piecewise linear function with steps **400**
16.3 A concave piecewise linear cost function **403**
16.4 A convex piecewise linear cost function **404**
19.1 Earliness and Tardiness as Piecewise Linear Cost Function **433**

List of Tables

1.1 Concert Technology Modeling Objects in C++ **49**
1.2 Optimizer Options in IloCplex::Algorithm **52**
1.3 Integer Parameters Tied to Nested Enumerations **53**
1.4 Algorithm Status and Information About the Model. **55**
2.1 Modeling Classes of ILOG CPLEX with Concert Technology for Java Users. **76**
2.2 Solution Status **81**
2.3 Algorithm Types for RootAlg **85**
2.4 Algorithm Types for NodeAlg **86**
2.5 Constants in IloCplex.PrimalPricing **87**
2.6 Classes with Parameters Defined by Integers. **87**
4.1 Special Data Types in the ILOG CPLEX Callable Library. **119**
4.2 Default Settings of ILOG CPLEX Growth Parameters **120**
4.3 Callable Library Routines for Parameters in the ILOG CPLEX Environment **122**
6.1 Options for the convert Utility and Corresponding File Extensions. **159**
6.2 Options for the Output-Channel Command. **162**
6.3 Channels Directing Output to Screen or to a File **163**
8.1 Settings of the LPMethod Parameter for Choosing an Optimizer **184**
8.2 Symbolic Names for LP Solution Methods **185**
8.3 Dependency Checking Parameter DepInd or CPX_PARAM_DEPIND. **189**
8.4 DPriInd Parameter Settings for Dual Simplex Pricing Algorithm. **192**
8.5 PPriInd Parameter Settings for Primal Simplex Pricing Algorithm **192**
8.6 ScaIndParameter Settings for Scaling Methods **193**
8.7 CraInd Parameter Settings for the Dual Simplex Optimizer **194**

8.8 CraInd Parameter Settings for the Primal Simplex Optimizer **194**
8.9 FeasOpt **210**
9.1 Settings of LPMethod Invoke the Barrier Optimizer **228**
9.2 Calling the Barrier Optimizer **229**
9.3 BarCrossAlg Parameter Settings **229**
9.4 BarDisplay Parameter Settings **230**
9.5 Infeasibilities and Norms in the Log File of a Barrier Optimization **232**
9.6 Barrier Solution Quality Display **233**
9.7 Dependency Checking Parameter DepInd or CPX_PARAM_DEPIND **237**
9.8 BarStartAlg Parameter Settings for Starting-Point Heuristics **239**
9.9 BarAlg Parameter Settings for Barrier Optimizer Algorithm **240**
11.1 RootAlg Parameter Settings **272**
13.1 Specifying Type of Variable in a MIP **307**
13.2 Interactive Optimizer Display Options for MIP Problems **308**
13.3 Problem Type Definitions **309**
13.4 Parameters for Controlling Branch & Cut Strategy **315**
13.5 NodeSel Parameter Settings for Node Search Type **316**
13.6 VarSel Parameter Settings for Branching Variable Choice **317**
13.7 BrDir Parameter Settings for Branching Direction Choice **317**
13.8 Parameters for Controlling Cuts **320**
13.9 Parameters for Controlling MIP Preprocessing **323**
13.10 Parameters for Branching Priority Order **325**
13.11 Parameters to Limit MIP Optimization **326**
13.12 Settings of the MIP Display Parameter **332**
13.13 Settings for the Node File Storage Parameter **342**
13.14 Settings of RootAlg and NodeAlg Parameters **345**
22.1 Callback Macros **481**
22.2 Status of Nonzero Callbacks for MIPs **487**
27.1 Thread-Limit Parameters **532**
27.2 Parallel Optimizer Commands in the Interactive Optimizer **535**
27.3 Parallel Optimizer Methods and Routines of Component Libraries **535**

List of Code Samples

ilodiet.cpp 67
Diet.java 94
diet.c 132
lpex5.c 165
Example of an infeasible.iis file 206
Example of infeasible.iis with many rows and columns 208
Time-limited search for an IIS 209
ilolpex6.cpp 216
lpex6.c 218
netex1.c 252
netex2.c 260
iloqpex1.cpp 274
QPex1.java 276
qpex1.c 278
qpex2.c 286
ilomipex1.cpp 346
mipex1.c 348
ilomipex2.cpp 356
mipex2.c 358
ilomipex3.cpp 362
mipex3.c 364
lpex7.c 444

ilogoalex1.cpp 456
ilogoalex2.cpp 463
ilogoalex3.cpp 471
ilolpex4.cpp 484
lpex4.c 489
iloadmipex5.cpp 501

Meet ILOG CPLEX

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

This preface introduces the *ILOG CPLEX User's Manual*. The manual assumes that you are familiar with ILOG CPLEX from reading *Getting Started with ILOG CPLEX* and from following the tutorials there. This preface covers these topics:

- *What Is ILOG CPLEX?* on page 26
- *What Does ILOG CPLEX Do?* on page 26
- *What You Need to Know* on page 28
- *In This Manual* on page 28
- *Related Documentation* on page 34
- *Further Reading* on page 36

What Is ILOG CPLEX?

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

- The **ILOG 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.
- **ILOG Concert Technology** is a set of libraries offering an API that includes modeling facilities to allow a programmer to embed ILOG CPLEX optimizers in C++, Java, or C#.NET applications. The library is provided in files `ilocplex.lib`, `concert.lib` and `cplex.jar` as well as `ILOG.CPLEX.dll` and `ILOG.CONCERT.dll` on Windows platforms and in `libilocplex.a`, `libconcert.a` and `cplex.jar` on UNIX platforms, and makes use of the Callable Library (described next).
- The **ILOG CPLEX Callable Library** is a C library that allows the programmer to embed ILOG CPLEX optimizers in applications written in C, Visual Basic, Fortran or any other language that can call C functions. The library is provided as a DLL on Windows platforms and in a library (that is, with file extensions `.a`, `.so`, or `.sl`) on UNIX platforms.

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

What Does ILOG CPLEX Do?

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

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

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

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

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

In the most basic linear optimization problem, the variables of the objective function are *continuous* in the mathematical sense, with no gaps between real values. To solve such linear programming problems, ILOG CPLEX implements optimizers based on the simplex algorithms (both primal and dual simplex) as well as primal-dual logarithmic barrier algorithms and a sifting algorithm. These alternatives are explained more fully in Chapter 8, *Solving LPs: Simplex Optimizers*.

ILOG CPLEX can also handle certain problems in which the objective function is not linear but *quadratic*. Such problems are known as *quadratic programs* or QPs. Chapter 11, *Solving Problems with a Quadratic Objective (QP)*, covers those kinds of problems.

ILOG CPLEX also solves certain kinds of quadratically constrained problems. Such problems are known as quadratically constrained programs or QCPs. Chapter 12, *Solving Problems with Quadratic Constraints (QCP)*, tells you more about the kinds of quadratically constrained problems that ILOG CPLEX solves.

ILOG CPLEX is also a tool for solving mathematical programming problems in which some or all of the variables must assume *integer* values in the solution. Such problems are known as *mixed integer programs* or MIPs because they may combine continuous and discrete (for example, integer) variables in the objective function and constraints. MIPs with linear objectives are referred to as *mixed integer linear programs* or MILPs, and MIPs with quadratic objective terms are referred to as *mixed integer quadratic programs* or MIQPs. Likewise, MIPs that are also quadratically constrained in the sense of QCP are known as *mixed integer quadratically constrained programs* or MIQCPs.

Within the category of mixed integer programs, there are two kinds of discrete integer variables: if the integer values of the discrete variables must be either 0 (zero) or 1 (one), then they are known as *binary*; if the integer values are not restricted in that way, they are known as general integer variables. This manual explains more about the mixed integer optimizer in Chapter 13, *Solving Mixed Integer Programming Problems (MIP)*.

ILOG CPLEX also offers a Network Optimizer aimed at a special class of linear problem with network structures. ILOG CPLEX can optimize such problems as ordinary linear programs, but if ILOG CPLEX can extract all or part of the problem as a network, then it will apply its more efficient Network Optimizer to that part of your problem and use the

partial solution it finds there to construct an advanced starting point to optimize the rest of the problem. Chapter 10, *Solving Network-Flow Problems* offers more detail about how the ILOG CPLEX Network Optimizer works.

What You Need to Know

Before you begin using ILOG CPLEX, it is a good idea to begin by reading *Getting Started with ILOG CPLEX* and to try the tutorials in it. It is available in the standard distribution of the product.

In order to use ILOG CPLEX effectively, you need to be familiar with your operating system, whether UNIX or Windows. A list of the machine-types and library formats (including version numbers of compilers and JDKs) is available in the standard distribution of your product in the file *yourCPLEXinstallation*`/mptable.html`.

This manual assumes that you are familiar with the concepts of mathematical programming, particularly linear programming. In case those concepts are new to you, the bibliography in *Further Reading* on page 36 in this preface indicates references to help you there.

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

In This Manual

This manual consists of these parts:

- Part I, Languages and APIs

  This part collects chapters about each of the application programming interfaces (APIs) available for ILOG CPLEX. It is not necessary to read each of these chapters thoroughly. In fact, most users will concentrate only on the chapter about the API that they plan to use, whether C, C++, Java, C#.NET, and so forth.

- Part II, Programming Considerations

  This part documents concepts that are valid as you develop an application, regardless of the programming language that you choose. It highlights software engineering concepts implemented in ILOG CPLEX, concepts that will enable you to develop effective applications to exploit it efficiently.

- Part III, Continuous Optimization

  This part focuses on algorithmic considerations about the optimizers of ILOG CPLEX that solve problems formulated in terms of *continuous* variables. While ILOG CPLEX is delivered with default settings that enable you to solve many problems without changing parameters, these chapters also document features that you can customize for your application.

- Part IV, Discrete Optimization

  This part focuses on algorithmic considerations about the optimizers of ILOG CPLEX that solve problems formulated in terms of *discrete* variables, such as integer, Boolean, piecewise-linear, or semi-continuous variables. Again, though default settings of ILOG CPLEX enable you to solve many problems without changing parameters, these chapters also document features that enable you to tune performance.

- Part V, Advanced Programming Techniques

  This part documents advanced programming techniques for users of ILOG CPLEX. It shows you how to apply query routines to gather information while ILOG CPLEX is working. It demonstrates how to redirect the search with goals or callbacks. This part also covers pools of user-defined cuts and pools of lazy constraints. It documents the advanced MIP control interface and the advanced aspects of preprocessing: presolve and aggregation. It also introduces special considerations about parallel programming with ILOG CPLEX. This part of the manual assumes that you are already familiar with earlier parts of the manual.

Part I, *Languages and APIs*

Chapter 1, *ILOG Concert Technology for C++ Users*, introduces Concert Technology. It provides an overview of the design of the library, explains modeling techniques, and offers an example of programming with Concert Technology. It also provides information about controlling parameters.

Chapter 2, *ILOG Concert Technology for Java Users*, explores the full range of features that the ILOG CPLEX Java API offers to solve mathematical programming problems. An overview of the architecture is given, then techniques for creating models are explained through examples.

Chapter 3, *ILOG Concert Technology for C#.NET Users*, offers an example of this API.

Chapter 4, *The ILOG CPLEX Callable Library*, introduces the ILOG CPLEX Callable Library. It sketches the architecture of the product, explains the relation between the Interactive Optimizer and the Callable Library, and offers an example of programming with the Callable Library. It also provides an overview about the parameters you control in ILOG CPLEX.

Part II, *Programming Considerations*

Chapter 5, *Developing CPLEX Applications*, provides tips for developing applications with ILOG CPLEX, suggests ways to debug your applications built around ILOG CPLEX, and provides a checklist to help avoid common programming errors.

Chapter 6, *Managing Input and Output*, explains how to enter mathematical programs efficiently and how to generate meaningful output from your ILOG CPLEX applications. It also lists the available file formats for entering data into ILOG CPLEX and writing bases and solutions from ILOG CPLEX.

Chapter 7, *Licensing an Application*, tells you what you must consider when you want to license your ILOG CPLEX application for deployment.

Part III, *Continuous Optimization*

Chapter 8, *Solving LPs: Simplex Optimizers*, goes deeper into aspects of linear programming with ILOG CPLEX. It explains how to tune performance and how to diagnose infeasibility in a model. It also offers an example showing you how to start optimizing from an advanced basis.

Chapter 9, *Solving LPs: Barrier Optimizer*, continues the exploration of optimizers for linear programming problems. It tells how to use the primal-dual logarithmic barrier algorithm implemented in the ILOG CPLEX Barrier Optimizer to solve large, sparse linear programming problems.

Chapter 10, *Solving Network-Flow Problems*, shows how to use the ILOG CPLEX Network Optimizer on linear programming problems based on a network model.

Chapter 11, *Solving Problems with a Quadratic Objective (QP)*, takes up programming problems in which the objective function may be quadratic. It, too, includes examples.

Chapter 12, *Solving Problems with Quadratic Constraints (QCP)*, introduces problems where the constraints are not strictly linear but may also include convex quadratic contraints and shows how to use the barrier optimizer to solve them.

Part IV, *Discrete Optimization*

Chapter 13, *Solving Mixed Integer Programming Problems (MIP)*, shows you how to handle MIPs. It particularly emphasizes performance tuning and offers a series of examples.

Chapter 14, *Cutting Stock: Column Generation*, takes a familiar example to explain how to build the model of a problem through column generation.

Chapter 15, *Rates: Using Semi-Continuous Variables*, uses a typical industrial problem to introduce semi-continous variables in a model.

Chapter 16, *Transport: Piecewise Linear Optimization*, another typical industrial problem, demonstrates how to represent parts of your problem as piecewise linear.

Chapter 17, *Logical Constraints in Optimization*, introduces logical constraints, such as logical-and, logical-or, if-then conditions, and so forth, that can be transformed automatically by ILOG CPLEX with Concert Technology for C++ users.

Chapter 18, *Food Manufacturing 2: Using Logical Constraints*, uses a problem formulation by H.P. Williams to show you how to use the ability of ILOG CPLEX to transform parts of your problem for you.

Chapter 19, *Early Tardy Scheduling*, shows you how to model and solve a scheduling problem that entails penalties for early or late completion of jobs consisting of activities assigned to resources.

Part V, *Advanced Programming Techniques*

Chapter 20, *Using Query Routines in the Callable Library*, shows how to access information about the model you currently have in memory through query routines of the Callable Library.

Chapter 21, *Using Goals*, shows how to use goals to control a MIP search.

Chapter 22, *Using Callbacks* shows how to use callbacks to control a MIP search.

Chapter 23, *Goals and Callbacks: a Comparison*, compares the two different approaches.

Chapter 24, *User Cut and Lazy Constraint Pools*, formerly available only through Technical Support, is now part of the standard documentation. It explains in greater detail how to manage your own pools of cuts and lazy constraints.

Chapter 25, *Advanced MIP Control Interface*, formerly available only through Technical Support, is now part of the standard documentation. It shows you how to exploit advanced features of MIP. It provides important additional information if you are using callbacks in your application.

Chapter 26, *Advanced Presolve Routines*, formerly available only through Technical Support, is now part of the standard documentation. It documents advanced aspects of presolve and aggregation more fully.

Chapter 27, *Parallel Optimizers*, explains how to exploit parallel optimizers in case your hardware supports parallel execution.

The *Index* on page 541 completes this manual.

Examples Online

For the examples explained in the manual, you will also see the complete code for the solution, so that you can see exactly how ILOG CPLEX fits into your own applications. In case you prefer to study code online, you'll also find the complete code for these examples in a subdirectory of the standard distribution of ILOG CPLEX.

The following table lists the examples in this manual and indicates where to find them, both online and in the manual:

Example	Source File	In This Manual
dietary optimization: building a model by rows (constraints) or by columns (variables), solving with `IloCplex` in C++	`ilodiet.cpp`	*Example: Optimizing the Diet Problem in C++* on page 61
dietary optimization: building a model by rows (constraints) or by columns (variables), solving with `IloCplex` in Java	`Diet.java`	*Example: Diet.java Source Code* on page 94
dietary optimization: building a model by rows (constraints) or by columns (variables), solving with `Cplex` in C#.NET	`Diet.cs`	*Example: Optimizing the Diet Problem in C#.NET* on page 111
dietary optimization: building a model by rows (constraints) or by columns (variables), solving with the Callable Library	`diet.c`	*Example: Optimizing the Diet Problem in the Callable Library* on page 129
linear programming: starting from an advanced basis	`ilolpex6.cpp` `lpex6.c`	*Example ilolpex6.cpp* on page 216 *Example lpex6.c* on page 218
network optimization: using the Callable Library	`netex1.c`	*Complete example* on page 252
network optimization: relaxing a network flow to an LP	`netex2.c`	*Complete example* on page 260

quadratic programming: maximizing a QP	`iloqpex1.cpp`
	`QPex1.java`
	`qpex1.c`
		Example: iloqpex1.cpp on page 274
		Complete Program: QPex1.java on page 276
		Complete Program: qpex1.c on page 278

quadratic programming: reading a QP from a formatted file	`qpex2.c`	*Complete example* on page 286

quadratically constrained programming: QCP	`qcpex1.c`
	`iloqcpex1.cpp`
	`QCPex1.java`
		Examples: QCP on page 301

mixed integer programming: optimizing a basic MIP	`ilomipex1.cpp` `mipex1.c`	*Complete Program: ilomipex1.cpp* on page 346 *Complete Program: mipex1.c* on page 347
mixed integer programming: reading a MIP from a formatted file	`ilomipex2.cpp` `mipex2.c`	*Complete Program: ilomipex2.cpp* on page 356 *Complete Program: mipex2.c* on page 357
mixed integer programming: using special ordered sets (SOS) and priority orders	`ilomipex3.cpp` `mipex3.c`	*Complete Program: ilomipex3.cpp* on page 362 *Complete Program: mipex3.c* on page 364
transport: piecewise-linear optimization	`transport.cpp`	*Complete Program: transport.cpp* on page 408

Example	Source File	In This Manual
food manufacturing 2: using logical constraints	`foodmanufac.cpp`	*Example: foodmanufact.cpp* on page 425
early tardy scheduling	`etsp.cpp`	*Example: etsp.cpp* on page 434
input and output: using the message handler	`lpex5.c`	*Complete Program: lpex5.c* on page 165
using query routines	`lpex7.c`	*Example: Using Query Routines lpex7.c* on page 443

using callbacks	`ilolpex4.c`
	`lpex4.c`
	`iloadmipex5.cpp`
		Complete example on page 484
		Complete example on page 489
		Complete example on page 501

Notation in This Manual

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

- Important ideas are *italicized* the first time they appear.
- The names of C routines and parameters in the ILOG CPLEX Callable Library begin with `CPX`; the names of C++ and Java classes in ILOG CPLEX Concert Technology begin with `Ilo`; and both appear in `this typeface`, for example: `CPXcopyobjnames` or `IloCplex`.
- The names of C#.NET classes and interfaces are the same as the corresponding entity in Java, except the name is **not** prefixed by `Ilo`. Names of C#.NET methods are the same as Java methods, except the C#.NET name is capitalized (that is, uppercase) to conform to Microsoft naming conventions.
- Where use of a specific language (C++, Java, C, C#.NET, and so on) is unimportant and the effect on the optimization algorithms is emphasized, the names of ILOG CPLEX parameters are given as their Concert Technology variant. The reference manual *ILOG CPLEX Parameters* contains a table showing the correspondence of these names to the Callable Library and the Interactive Optimizer.
- Text that is entered at the keyboard or displayed on the screen and commands and their options available through the Interactive Optimizer appear in `this typeface`, for example, `set preprocessing aggregator n`.
- Values that you must fill in (for example, the value to set a parameter) also appear in the same typeface as the command but modified to indicate you must supply an appropriate value; for example, `set simplex refactor` *`i`* indicates that you must fill in a value for *`i`*.
- Matrices are denoted in two ways:

- In printable material where superscripts and bold type are available, the product of A and its transpose is denoted like this: $\boldsymbol{AA}^{\mathrm{T}}$. The superscript T indicates the matrix transpose.
- In computer-generated samples, such as log files, where only ASCII characters are available, the product of A and its transpose are denoted like this: `A*A'`. The asterisk (*) indicates matrix multiplication, and the prime (') indicates the matrix transpose.

Related Documentation

The online information files are distributed with the ILOG CPLEX libraries and can be found in *yourCplexHome*`/doc` and its subdirectories:

- `html`
- `pdf`
- `windows`

The complete documentation set for ILOG CPLEX consists of the following material:

- **ILOG CPLEX Getting Started:** It is a good idea for new users of ILOG CPLEX to start with that manual. It introduces ILOG CPLEX through the Interactive Optimizer, and contains tutorials for ILOG CPLEX Concert Technology for C++, Java, and C#.NET applications as well as the ILOG CPLEX Callable Library.

  The *ILOG CPLEX Getting Started* manual is supplied in HTML, in Microsoft compiled HTML help (.chm), and as a PDF file.

- **ILOG CPLEX User's Manual:** This manual explains the topics covered in the *Getting Started* manual in greater depth, with individual chapters about:

  - LP (Linear Programming) problems;
  - Network-Flow problems;
  - QP (Quadratic Programming) problems;
  - QCP (Quadratically Constrained Programming), and
  - MIP (Mixed Integer Programming) problems.

  There is also detailed information about:

  - managing input and output,
  - using query routines,
  - using callbacks, and
  - using parallel optimizers.

The *ILOG CPLEX User's Manual* is supplied in HTML form, in Microsoft compiled HTML help (.chm), and as a PDF file.

- **ILOG CPLEX Callable Library and C++ API Reference Manual:** This manual supplies detailed definitions of the routines, classes, macros, and functions in the ILOG CPLEX Callable Library C and C++ application programming interface (API). It is available online as HTML, as Microsoft compiled HTML help (.chm), and as a PDF file.

  As part of that online manual, you can also access other reference material:

  - **Overview of the API** offers you navigational links into the HTML reference manual organized into categories of tasks you may want to perform in your applications. Each category includes a table linking to the corresponding C routine, C++ class or method, and Java interface, class, or method to accomplish the task. There are also indications about the name of the corresponding C#.NET method so you can locate it in the Microsoft compiled HTML help (.chm).
  - **ILOG CPLEX Error Codes** documents error codes by name in the group `optim.cplex.errorcodes`. You can also acess error codes by number in the *Overview of the API* through the link *Interpreting Error Codes*.
  - **ILOG CPLEX Solution Quality Codes** documents solution quality codes by name in the group `optim.cplex.solutionquality`.
  - **ILOG CPLEX Solution Status Codes** documents solution status codes by name in the group `optim.cplex.solutionstatus`. You can also acess solution status codes by number in the *Overview of the API* through the link *Interpreting Solution Status Codes*.

- **ILOG Concert C++ API Reference Manual:** This manual supplements the ILOG CPLEX C++ API Reference Manual with more information about the modeling layer.

- **ILOG CPLEX Parameters Reference Manual**: This manual lists the parameters of ILOG CPLEX with their names in the Callable Library, in Concert Technology, and in the Interactive Optimizer. It also shows their default settings with explanations of the effect of other settings. Normally, the default settings of ILOG CPLEX solve a wide range of mathematical programming problems without intervention on your part, but these parameters are available for fine tuning in special cases.

- **ILOG CPLEX File Formats Reference Manual**: This manual documents the file formats recognized and supported by ILOG CPLEX.

- **ILOG CPLEX Interactive Optimizer Reference Manual**: This manual lists the commands of the Interactive Optimizer, along with their options and links to examples of their use in the *ILOG CPLEX User's Manual*.

- **ILOG CPLEX C#.NET Reference Manual:** This manual documents the C#.NET API of Concert Technology for ILOG CPLEX. It is available as Microsoft compiled HTML help (.chm).
- **ILOG CPLEX Java Reference Manual:** This manual supplies detailed definitions of the Concert Technology interfaces and ILOG CPLEX Java classes. It is available online as HTML and as Microsoft compiled HTML help (.chm).
- **ILOG License Manager (ILM):** ILOG products are protected by the ILOG License Manager. Before you can use ILOG CPLEX, you need to set up ILM. Its online documentation explains how to do so step-by-step, for different platforms. It is in HTML form, included with your distribution.

Announcements: cplex-list@ilog.fr

The electronic mailing list `cplex-list@ilog.fr` is available to keep you informed about important product updates. If you subscribe to this list, you will receive announcements when new releases are available, updates to FAQs and code samples, or possibly an invitation to beta testing.

To subscribe to this list, go to the ILOG Customer Support web site and navigate to the ILOG CPLEX product support pages in the Products section. The link *Users List* enables you access a page where you can subscribe to the ILOG CPLEX mailing list.

Only the product manager of ILOG CPLEX posts announcements to this list. Your name and mailing address will not be published for any other purpose than receiving these official announcements.

Further Reading

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

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

Wolsey, Laurence A., *Integer Programming*, New York: John Wiley & Sons, 1998. This book explains branch and cut, including cutting planes, in detail.

Nemhauser, George L. and Laurence A. Wolsey, *Integer and Combinatorial Optimization*, New York: John Wiley & Sons, 1999. A reprint of the 1988 edition. This book is a widely cited and comprehensive reference about integer programming.

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

Part I

Languages and APIs

This part of the manual collects chapters about each of the application programming interfaces (APIs) available for ILOG CPLEX. It is not necessary to read each of these chapters thoroughly. In fact, most users will concentrate only on the chapter about the API that they plan to use, whether C, C++, Java, or C#.NET. This part contains:

- *ILOG Concert Technology for C++ Users* on page 41
- *ILOG Concert Technology for Java Users* on page 73
- *ILOG Concert Technology for C#.NET Users* on page 101
- *The ILOG CPLEX Callable Library* on page 113

1

ILOG Concert Technology for C++ Users

This chapter shows how to write C++ programs using ILOG CPLEX Concert Technology for C++ users. It includes sections about:

- *The Design of ILOG CPLEX Concert Technology* on page 41, including information about licensing, compiling, and linking your programs
- *Creating an Application with Concert Technology* on page 43
- *Modeling an Optimization Problem with Concert Technology* on page 43
- *Solving Concert Technology Models with IloCplex* on page 47
- *Accessing Solution Information* on page 54
- *Modifying a Model* on page 58
- *Handling Errors* on page 60
- *Example: Optimizing the Diet Problem in C++* on page 61

The Design of ILOG CPLEX Concert Technology

Figure 1.1, *A View of Concert Technology for C++ Users* shows a program using ILOG CPLEX Concert Technology to solve optimization problems. The optimization part of

the user's application program is captured in a set of interacting C++ objects that the application creates and controls. These objects can be divided into two categories:

1. **Modeling objects** are used to define the optimization problem. Generally an application creates several modeling objects to specify the optimization problems. Those objects are grouped into an `IloModel` object representing the complete optimization problem.

2. **`IloCplex` objects** are used to solve models created with the modeling objects. An `IloCplex` object reads a model and extracts its data to the appropriate representation for the ILOG CPLEX optimizer. Then the `IloCplex` object is ready to solve the model it extracted and be queried for solution information.

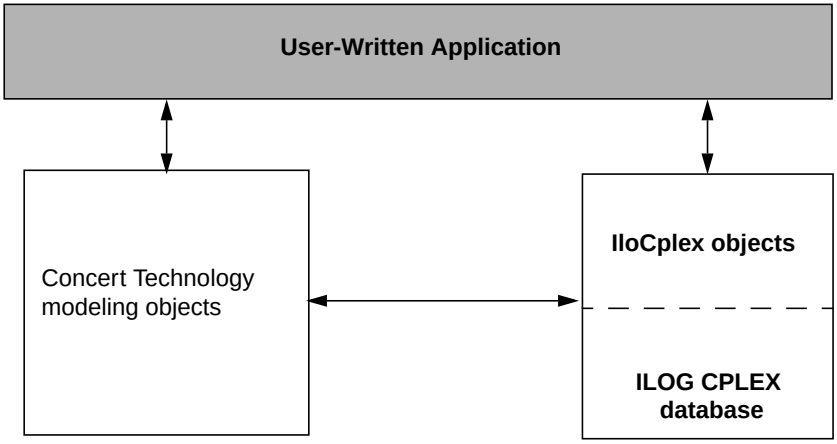

Figure 1.1 *A View of Concert Technology for C++ Users*

Licenses

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

Compiling and Linking

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

Creating an Application with Concert Technology

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

- First, create a model of your problem with the modeling facilities of Concert Technology. *Modeling an Optimization Problem with Concert Technology* on page 43 offers an introduction to creating a model.
- When the model is ready to be solved, hand it over to ILOG CPLEX for solving. *Solving Concert Technology Models with IloCplex* on page 47 explains how to do so. It includes a survey of the `IloCplex` interface for controlling the optimization. Individual controls are discussed in the chapters explaining the individual optimizers.
- *Accessing Solution Information* on page 54, shows you how to access and interpret results from the optimization after solving the model.
- After analyzing the results, you may make changes to the model and study their effect. *Modifying a Model* on page 58 explains how to make changes and how ILOG CPLEX deals with them.
- *Handling Errors* on page 60, discusses the error handling and debugging support provided by Concert Technology and ILOG CPLEX.
- *Example: Optimizing the Diet Problem in C++* on page 61 presents a complete program.

Not covered in this chapter are advanced features, such as the use of goals or callbacks for querying data about an ongoing optimization and for controlling the optimization itself. Goals, callbacks, and other advanced features are discussed in Part V, *Advanced Programming Techniques*.

Modeling an Optimization Problem with Concert Technology

This section briefly introduces Concert Technology for modeling optimization problems to be solved by `IloCplex`. It highlights these topics:

- *Modeling Classes* on page 44
  - *Creating the Environment: IloEnv* on page 44
  - *Defining Variables and Expressions: IloNumVar* on page 44
  - *Declaring the Objective: IloObjective* on page 45
  - *Adding Constraints: IloConstraint and IloRange* on page 46
  - *Formulating a Problem: IloModel* on page 46
- *Data Management Classes* on page 47

Modeling Classes

A Concert Technology model consists of a set of C++ objects. Each variable, each constraint, each special ordered set (SOS), and the objective function in a model are all represented by objects of the appropriate Concert Technology class. These objects are known as modeling objects.

Creating the Environment: IloEnv

Before you create modeling objects, you must construct an object of the class `IloEnv`. This object known as the environment. It is constructed with the statement:

```
IloEnv env;
```

That statement is usually the first Concert Technology statement in an application. At the end, you must close the environment by calling:

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

That statement is usually the last Concert Technology statement in an application. The `end` method must be called because, like most Concert Technology classes, the class `IloEnv` is a handle class. That is, an `IloEnv` object is really only a pointer to an implementation object. Implementation objects are destroyed by calling the `end` method. Failing to call the `end` method can result in memory leaks.

Users familiar with the ILOG CPLEX Callable Library are cautioned not to confuse the Concert Technology environment object with the ILOG CPLEX environment object of type `CPXENVptr`, used for setting ILOG CPLEX parameters. Such an object is not needed with Concert Technology, as parameters are handled directly by each instance of the class `IloCplex`. In other words, the environment in Concert Technology always refers to the object of class `IloEnv` required for all other Concert Technology objects.

Defining Variables and Expressions: IloNumVar

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

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

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

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

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

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

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

Whenever possible, build your expressions in terms of data that is either integer or double-precision (64-bit) floating point. Single-precision (32-bit) floating point data should be avoided, as it can result in unnecessarily ill conditioned problems. For more information, refer to *Numerical Difficulties* on page 196.

While Concert Technology supports very general expressions, only linear, quadratic, piecewise-linear, and logical expressions can be used in models to be solved with `IloCplex`. For more about each of those possibilities, see these chapters of this manual:

- Chapter 8, *Solving LPs: Simplex Optimizers* and Chapter 9, *Solving LPs: Barrier Optimizer* both discuss linear expressions.
- Chapter 11, *Solving Problems with a Quadratic Objective (QP)* discusses quadratic expressions in an objective function.
- Chapter 12, *Solving Problems with Quadratic Constraints (QCP)* discusses quadratic expressions in quadratically constrained programming problems (QCP).
- Chapter 16, *Transport: Piecewise Linear Optimization* introduces piecewise-linear expressions through a transportation example.
- Chapter 17, *Logical Constraints in Optimization* introduces logical constraints handled by ILOG CPLEX. Chapters following it offer examples.

When you have finished using an expression (that is, you created a constraint with it) you need to delete it by calling its method `end`, for example:

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

Declaring the Objective: IloObjective

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

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

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

Adding Constraints: IloConstraint and IloRange

Similarly, objects of the class `IloConstraint` represents constraints in your model. Most constraints will belong to the subclass `IloRange`, derived from `IloConstraint`, and thus inherit its constructors and methods. `IloRange` represent constraints of the form `lower bound` ≤ `expression` ≤ `upper bound`. In other words, an instance of `IloRange` is a convenient way to express a ranged constraint, that is, a constraint with explicit upper or lower bounds. Any floating-point value or `+IloInfinity` or `-IloInfinity` can be used for the bounds. For example:

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

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

Formulating a Problem: IloModel

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

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

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

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

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

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

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

Data Management Classes

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

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

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

produces the following output:

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

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

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

```
cin >> data;
```

The example at the end of this chapter (*Example: Optimizing the Diet Problem in C++* on page 61) takes advantage of this function and reads the problem data from a file.

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

Solving Concert Technology Models with IloCplex

ILOG CPLEX generally does not need to be involved while you create your model. However, after the model is set up, it is time to create your `cplex` object, that is, an instance of the class `IloCplex`, to be used to solve the model. `IloCplex` is a class derived from

`IloAlgorithm`. There are other Concert Technology algorithm classes, also derived from `IloAlgorithm`, as documented in the *Concert Technology Reference Manual*. Some models might also be solved by using other algorithms, such as the class `IloSolver` for constraint programming, or by using a hybrid algorithm consisting of both `IloSolver` and ILOG CPLEX. Some models, on the other hand, cannot be solved with ILOG CPLEX.

The makeup of the model determines whether or not ILOG CPLEX can be used to solve it. More precisely, in order to be handled by `IloCplex` objects, a model may only consist of modeling objects of the classes listed in Table 1.1.

Instances of `IloConstraint` extracted by ILOG CPLEX can be created in a variety of ways. Most often, they can be generated by means of overloaded C++ operators, such as `==`, `<=`, or `>=`, in the form `expression1 operator expression2`. Instances of both `IloConstraint` and `IloRange` generated in that way may be built from either linear or quadratic expressions. They may also include piecewise linear terms if variables can be substituted in such a way that the resulting expression is linear or quadratic.

For more detail about solving problems with `IloCplex`, see the following sections of this manual:

- *Extracting a Model* on page 49
- *Solving a Model* on page 50
- *Choosing an Optimizer* on page 51
- *Controlling the Optimizers* on page 52

Table 1.1 *Concert Technology Modeling Objects in C++*

To model:	Use:
numerical variables	objects of the class `IloNumVar`, as long as they are not constructed with a list of feasible values
semi-continuous variables	objects of the class `IloSemiContVar`
linear objective function	an object of the class `IloObjective` with linear or piecewise linear expressions
quadratic objective function	an object of the class `IloObjective` with quadratic expressions
linear constraints	objects of the class `IloRange` (lower bound <= expression <= upper bound) or objects of the class `IloConstraint` (expr1 relation expr2) involving strictly linear or piecewise linear expressions
quadratic constraints	objects of the class `IloConstraint` that contain quadratic expressions as well as linear expressions or piecewise linear expressions
logical constraints	objects of the class `IloConstraint` or generated ranges with linear or piecewise linear expressions
variable type-conversions	objects of the class `IloConversion`
special ordered sets of type 1	objects of the class `IloSOS1`
special ordered sets of type 2	objects of class `IloSOS2`

For an explanation of quadratic constraints, see *Solving Problems with Quadratic Constraints (QCP)* on page 295. For more information about quadratic objective functions, see *Solving Problems with a Quadratic Objective (QP)* on page 265. For examples of piecewise linear constraints, see *Transport: Piecewise Linear Optimization* on page 397. For more about logical constraints, see *Logical Constraints in Optimization* on page 411. For a description of special ordered sets, see *Using Special Ordered Sets* on page 329.

Extracting a Model

This manual defines only one optimization model and uses only one instance of `IloCplex` at a time to solve the model. Consequently, it talks about these as the model and the `cplex` object. It should be noted, however, that in Concert Technology an arbitrary number of models and algorithm-objects can be created. The `cplex` object can be created by the constructor:

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

To use it to solve the model, the model must first be extracted to `cplex` by a call like this:

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

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

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

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

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

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

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

Solving a Model

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

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

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

Choosing an Optimizer

Solving the extracted model with ILOG CPLEX involves solving one or a series of continuous relaxations:

- Only one continuous relaxation needs to be solved if the extracted model is continuous itself, that is, if it does not contain integer variables, Boolean variables, semi-continuous or semi-integer variables, logical constraints, special ordered sets (SOS), or piecewise linear functions. Chapter 8, *Solving LPs: Simplex Optimizers* and Chapter 9, *Solving LPs: Barrier Optimizer* discuss the algorithms available for solving LPs. Similarly, Chapter 11, *Solving Problems with a Quadratic Objective (QP)*, discusses the algorithms available for solving QPs. Chapter 12, *Solving Problems with Quadratic Constraints (QCP)* re-introduces the barrier optimizer in the context of quadratically constrained programming problems (QCPs). Chapter 16, *Transport: Piecewise Linear Optimization* introduces piecewise-linear functions through a transportation example. Chapter 17, *Logical Constraints in Optimization* introduces logical constraints, and chapters following it offer examples.
- In all other cases, the extracted problem that ILOG CPLEX sees is indeed a MIP and, in general, a *series* of continuous relaxations needs to be solved. The method `cplex.isMIP` returns `IloTrue` in such a case. Chapter 13, *Solving Mixed Integer Programming Problems (MIP)* discusses the algorithms applied.

The optimizer option used for solving the first continuous relaxation (whether it is the only one or just the first in a series of problems) is controlled by setting the root algorithm parameter:

```
cplex.setParam(IloCplex::RootAlg, alg);
```

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

```
enum IloCplex::Algorithm {
  NoAlg,
  AutoAlg,
  Primal,
  Dual,
  Barrier,
  Network,
  Sifting,
  Concurrent
};
```

The choice `Sifting` is **not** available for QP models. Only the `Barrier` option is available for QCP models.

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

Table 1.2 *Optimizer Options in IloCplex::Algorithm*

`IloCplex::Primal`	use the primal simplex algorithm
`IloCplex::Dual`	use the dual simplex algorithm
`IloCplex::Barrier`	use the barrier algorithm. The type of crossover performed after the barrier algorithm is determined by parameter `IloCplex::BarCrossAlg`.
`IloCplex::Network`	use the primal network simplex algorithm on an embedded network followed by the dual simplex algorithm for LPs and the primal simplex algorithm for QPs on the entire problem
`IloCplex::Sifting`	use the sifting algorithm
`IloCplex::Concurrent`	use multiple algorithms concurrently on a multiprocessor system

If the extracted model requires the solution of more than one continuous relaxation, the algorithm for solving all but the first is controlled by the `NodeAlg` parameter:

```
cplex.setParam(IloCplex::NodeAlg, alg).
```

Controlling the Optimizers

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

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

Some integer parameters are tied to nested enumerations that define symbolic constants for the values the parameter may assume. Table 1.3 summarizes those parameters and their enumeration types.

Table 1.3 *Integer Parameters Tied to Nested Enumerations*

This Enumeration Type:	Is Used for This Parameter:
`IloCplex::MIPEmphasisType`	`IloCplex::MIPEmphasis`
`IloCplex::VariableSelect`	`IloCplex::VarSel`
`IloCplex::NodeSelect`	`IloCplex::NodeSel`
`IloCplex::PrimalPricing`	`IloCplex::PPriInd`
`IloCplex::DualPricing`	`IloCplex::DPriInd`
`IloCplex::BranchDirection`	`IloCplex::BrDir`

Only the parameter `IloCplex::MIPEmphasis` may be of importance for general use.

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

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

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

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

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

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

When solving MIPs, additional controls of the solution process are provided. Priority orders and branching directions can be used to control the branching in a static way. These are discussed in *Heuristics* on page 321. These controls are static in the sense that they allow you to control the solution process based on data that does not change during the solution and can thus be set up before solving the model.

Dynamic control of the solution process of MIPs is provided through goals or control callbacks. They are discussed in *Using Goals* on page 453, and in *Using Callbacks* on page 477. Goals and callbacks allow you to control the solution process based on

information that is generated during the solution process. *Goals and Callbacks: a Comparison* on page 505 contrasts the advantages of each approach.

Accessing Solution Information

Information about solution feasibility, solution variables, basis information, and solution quality can be accessed with the methods documented in the following sections.

- *Accessing Solution Status* on page 54
- *Querying Solution Data* on page 55
- *Accessing Basis Information* on page 56
- *Performing Sensitivity Analysis* on page 56
- *Analyzing Infeasible Problems* on page 57
- *Solution Quality* on page 57

Accessing Solution Status

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

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

Notice that the fully qualified names have the `IloAlgorithm` prefix. Table 1.4 shows what each return status means for the extracted model.

Table 1.4 *Algorithm Status and Information About the Model*

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

As can be seen, these statuses indicate information about the model that the ILOG CPLEX optimizer was able to prove during the last call to method `solve`. In addition, the ILOG CPLEX optimizer provides information about how it terminated. For example, it may have terminated with only a feasible but not optimal solution because it hit a limit or because a user callback terminated the optimization. Further information is accessible by calling solution query routines, such as method `cplex.getCplexStatus`, which returns a member of the nested enumeration type `IloCplex::CplexStatus`, or methods `cplex.isPrimalFeasible` or `cplex.isDualFeasible`.

For more information about those status codes, see the *ILOG CPLEX Reference Manual*.

Querying Solution Data

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

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

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

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

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

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

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

Accessing Basis Information

When solving LPs or QPs with either the simplex algorithm or the barrier optimizer with crossover enabled, basis information is available as well. Basis information can be consulted by the method `IloCplex::getBasisStatuses` which returns basis status information for variables and constraints.

Such information is encoded by the nested enumeration:

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

Performing Sensitivity Analysis

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

Analyzing Infeasible Problems

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

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

When an LP has been proven to be infeasible, ILOG CPLEX provides assistance for determining the cause of the infeasibility. This is done by computing what is known as an irreducibly inconsistent set (IIS), which is a description of the minimal subproblem that is still infeasible. Here minimality is defined as being: if you remove any of the constraints (including finite bounds), the infeasibility vanishes. An IIS is computed for an infeasible LP model by calling method `cplex.getIIS`. To compute an IIS for a QP, you should change the model to an LP by removing (zeroing) any quadratic terms from the objective function and then compute an IIS on the LP version. IIS information is not available for QCPs.

Solution Quality

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

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

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

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

ILOG CPLEX will copy the variable or constraint handle in which the maximum violation occurs to `maxvar` or `maxrange` and make the other handle an empty one. The maximum primal infeasibility is only one example of a wealth of quality measures. The full list is

defined by the nested enumeration type `IloCplex::Quality`. All of these can be used as a parameter for the `getQuality` methods, though some measures are not available for all optimizer option choices. A list of solution qualities appears in the *ILOG CPLEX Reference Manual*, Callable Library and C++ API, as the group `optim.cplex.solutionquality`.

Modifying a Model

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

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

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

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

Deleting and Removing Modeling Objects

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

Concert Technology also provides a way to remove a modeling object from all other modeling objects and algorithms exactly the same way as when deleting it, yet without

deleting the modeling object. This is done by calling the method `removeFromAll`. This may be helpful to temporarily remove a variable from your model while keeping the option to add it back later on.

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

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

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

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

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

Changing Variable Type

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

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

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

Handling Errors

In Concert Technology two kinds of errors are distinguished:

1. Programming errors, such as:

   - accessing empty handle objects;
   - mixing modeling objects from different environments;
   - accessing Concert Technology array elements beyond an array's size; and
   - passing arrays of incompatible size to functions.

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

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

2. Runtime errors, such as memory exhaustion.

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

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

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

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

The error message can be queried by calling method:

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

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

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

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

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

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

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

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

Example: Optimizing the Diet Problem in C++

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

- *Creating a Model Row by Row* on page 62;
- *Creating a Model Column by Column* on page 62;
- *Creating Multi-Dimensional Arrays with IloArray* on page 63;
- *Using Arrays for Input/Output* on page 64;
- *Solving the Model with IloCplex* on page 66.

Problem Representation

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

- The problem can be modeled by rows, by entering the variables first and then adding the constraints on the variables and the objective function.
- The problem can be modeled by columns, by constructing a series of empty constraints and then inserting the variables into the constraints and the objective function.

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

Creating a Model Row by Row

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

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

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

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

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

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

Creating a Model Column by Column

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

`nutrMin[i] ≤ ... ≤ nutrMax[i]`

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

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

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

where the notation + and `sum` indicate that you add the new variable `j` to the objective `cost` and constraints range[`i`]. The value in parentheses is the linear coefficient that is used for the new variable. This notation is similar to the syntax actually used in Concert Technology, as demonstrated in the function `buildModelByColumn`, in example `ilodiet.cpp`.

Creating Multi-Dimensional Arrays with IloArray

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

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

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

Application Description

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

> ***Note:*** *In such cases, an exception is thrown. This practice makes sure that* `env.end` *is called before the program is terminated.*

Using Arrays for Input/Output

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

After the problem data has been read and verified, it is time to build the model. To do so, construct the model object with this declaration:

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

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

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

Building the Model by Row

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

The statement:

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

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

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

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

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

After an expression has been used for creating a constraint, it is deleted by a call to `expr.end`.

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

Building the Model by Column

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

The statement:

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

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

In the following loop, the variables of the model are created one by one in columns; thus, the new variables are immediately installed in the model. An `IloNumColumn` object `col` is created and initialized to define how each new variable will be appended to the existing objective and constraints.

The `IloNumColumn` object `col` is initialized to contain the objective coefficient for the new variable. This is created with `cost(foodCost[j])`, that is using the overloaded `operator()` for `IloObjective`. Next, an `IloNumColumn` object is created for every constraint, representing the coefficient the new variable has in that constraint. Again these `IloNumColumn` objects are created with the overloaded `operator()`, this time of `IloRange`. The `IloNumColumn` objects are merged together to an aggregate `IloNumColumn` object using operator `+=`. The coefficient for row `i` is created with

`range[i](nutrPer[i][j])`, which calls the overloaded `operator()` for `IloRange` objects.

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

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

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

Solving the Model with IloCplex

After the model has been populated, it is time to create the `cplex` object and extract the model to it by calling:

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

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

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

Complete Program

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

Notes:

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

```
// ------------------------------------------------------------ -*- C++ -*-
// File: examples/src/ilodiet.cpp
// Version 9.0
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// A dietary model.
//
// Input data:
// foodMin[j]      minimum amount of food j to use
// foodMax[j]      maximum amount of food j to use
// foodCost[j]     cost for one unit of food j
// nutrMin[i]      minimum amount of nutrient i
// nutrMax[i]      maximum amount of nutrient i
// nutrPer[i][j]   nutrition amount of nutrient i in food j
//
// Modeling variables:
// Buy[j]          amount of food j to purchase
//
// Objective:
// minimize sum(j) Buy[j] * foodCost[j]
//
// Constraints:
// forall foods i: nutrMin[i] <= sum(j) Buy[j] * nutrPer[i][j] <= nutrMax[j]
//

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN


void usage(const char* name) {
   cerr << endl;
```

```
   cerr << "usage:   " << name << " [options] <file>" << endl;
   cerr << "options: -c  build model by column" << endl;
   cerr << "         -i  use integer variables" << endl;
   cerr << endl;
}


typedef IloArray<IloNumArray> IloNumArray2;


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

   Buy.clear();
   IloNumVarArray tmp(env, foodMin, foodMax, type);
   Buy.add(tmp);
   tmp.end();

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

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


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

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

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

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


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

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

      IloInt i;

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

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

      // model data
```

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

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

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

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

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

      // Build model

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

      // Solve model

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

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

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

```
      cerr << "Error: " << ex << endl;
   }
   catch (...) {
      cerr << "Error" << endl;
   }

   env.end();

   return 0;
}
```

C H A P T E R

2

ILOG Concert Technology for Java Users

This chapter explores the features ILOG CPLEX offers to Java users to solve mathematical programming problems. The chapter offers an overview of the architecture, and then explains techniques for creating models with ranged constraints and for creating objective functions. These elements are then used to build and solve the diet problem, introduced in *Example: Optimizing the Diet Problem in C++* on page 61.

- *Architecture of a CPLEX Java Application* on page 74
- *Modeling with Concert Technology* on page 75
- *Building the Model* on page 79
- *Solving the Model* on page 81
- *Accessing Solution Information* on page 82
- *Choosing an Optimizer* on page 83
- *Controlling ILOG CPLEX Optimizers* on page 86
- *More Solution Information* on page 88
- *Advanced Modeling with IloLPMatrix* on page 91
- *Modeling by Column* on page 92
- *Optimizing the Diet Problem in Java* on page 93

Architecture of a CPLEX Java Application

A user-written application first creates an `IloCplex` object. It then uses the Concert Technology modeling interface implemented by `IloCplex` to create the variables, the constraints, and the objective function of the model to be solved. For example, every variable in a model is represented by an object that implements the Concert Technology variable interface `IloNumVar`. The user code only accesses the variable through its Concert Technology interface. Similarly, all other modeling objects are accessed only through their respective Concert Technology interfaces from the user-written application, while the actual objects are maintained in the ILOG CPLEX database.

Figure 2.1 illustrates how an application uses Concert Technology, `IloCplex`, and the ILOG CPLEX database. The Java interfaces, represented by the dashed outline, do not actually consume memory. The ILOG CPLEX database includes the computing environment, its communication channels, and your problem objects.

For users familiar with object-oriented design patterns, this design is that of a factory, where `IloCplex` is a factory for modeling objects. The advantage of such a design is that code which creates a model using the Concert Technology modeling interface can be used not only with `IloCplex`, but also with any other factory class, for instance `IloSolver`. This allows you to try different ILOG optimization technologies for solving your model.

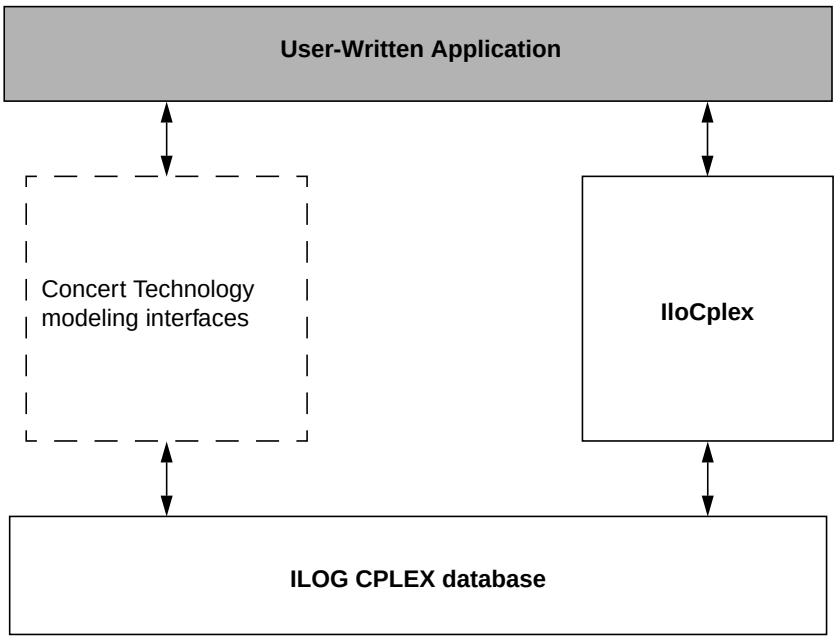

Figure 2.1 *A View of Concert Technology for Java Users*

Modeling with Concert Technology

An optimization problem is represented by a set of interconnected modeling objects in an `IloCplex` object. Modeling objects in Concert Technology are objects of type `IloNumVar` and its extensions, or `IloAddable` and its extensions. Since these are Java interfaces and not classes, objects of these types cannot be created explicitly. Rather, modeling objects are created using methods of an instance of `IloModeler` or one of its extensions, such as `IloMPModeler` or `IloCPModeler`.

This discussion concentrates on `IloModeler` and `IloMPModeler` because the class `IloCplex` implements these interfaces and thus inherits their methods. To create a new modeling object, you must first create the `IloModeler` which will be used to create the modeling object. For the discussion here, the model will be an instance of `IloCplex`, and it is created like this:

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

Since class `IloCplex` implements `IloMPModeler` (and thus its parent interface `IloModeler`) all methods from `IloMPModeler` and `IloModeler` can be used for building a model. `IloModeler` defines the methods to:

- create modeling variables of type integer, floating-point, or Boolean;
- construct simple expressions using modeling variables;
- create objective functions; and
- create ranged constraints, that is, constraints of the form:

  `lowerbound ≤ expression ≤ upperbound`

Models that consist only of such constructs can be built and solved with any ILOG optimizer implementing the `IloModeler` interface, including `IloCplex`, which implements the `IloMPModeler` extension.

The `IloMPModeler` interface extends `IloModeler` by adding functionality specific to mathematical programming applications. This functionality includes these additional modeling object types:

- semi-continuous variables;
- special ordered sets; and
- piecewise linear functions.

It also includes these modeling features to support specific needs:

- change of type for previously declared variables;
- modeling by column; and
- general manipulations of model entities.

Table 2.1 recapitulates those observations about the interfaces of ILOG CPLEX with Concert Technology for Java users.

Table 2.1 *Modeling Classes of ILOG CPLEX with Concert Technology for Java Users*

To Model This	Use an Object of This Class
variable	`IloNumVar` and its extensions `IloIntVar` and `IloSemiContVar`
range constraint	`IloRange` with (piecewise) linear or quadratic expressions
other relational constraint	`IloConstraint` of the form *expr1 relation expr2*, where both expressions are linear or quadratic and may optionally contain piecewise linear terms.
LP matrix	`IloLPMatrix`
linear or quadratic objective	`IloObjective` with (piecewise) linear or quadratic expressions
variable type-conversion	`IloConversion`
special ordered set	`IloSOS1` or `IloSOS2`

For an explanation of quadratic constraints, see *Solving Problems with Quadratic Constraints (QCP)* on page 295. For more information about quadratic objective functions, see *Solving Problems with a Quadratic Objective (QP)* on page 265. For examples of piecewise linear constraints, see *Transport: Piecewise Linear Optimization* on page 397. For a description of special ordered sets, see *Using Special Ordered Sets* on page 329.

Modeling with IloModeler

`IloModeler` defines an interface for building optimization models. This interface defines methods for constructing variable, constraint, and objective function objects.

Modeling Variables

A modeling variable in Concert Technology is represented by an object of type `IloNumVar` or one of its extensions. You can choose from a variety of methods defined in `IloModeler` and `IloMPModeler` to create one or multiple modeling variable objects. An example of the method is:

```
IloNumVar x = cplex.numVar(lb, ub, IloNumVarType.Float, "xname");
```

This constructor method allows you to set all the attributes of a variable: its lower and upper bounds, its type, and its name. Names are optional in the sense that `null` strings are considered to be valid as well.

The other constructor methods for variables are provided mainly for ease of use. For example, because names are not frequently assigned to variables, all variable constructors come in pairs, where one variant requires a name string as the last parameter and the other one does not (defaulting to a `null` string).

Integer variables can be created by the `intVar` methods, and do not require the type `IloNumVarType.Int` to be passed, as this is implied by the method name. The bound parameters are also specified more consistently as integers. These methods return objects of type `IloIntVar`, an extension of interface `IloNumVar` that allows you to query and set bounds consistently using integers, rather than doubles as used for `IloNumVar`.

Frequently, integer variables with `0/1` bounds are used as decision variables. To help create such variables, the `boolVar` methods are provided. In the Boolean type, `0` (zero) and `1` (one) are implied, so these methods do not need to accept any bound values.

For all these constructive methods, there are also equivalent methods for creating a complete array of modeling variables at one time. These methods are called `numVarArray`, `intVarArray`, and `boolVarArray`.

Building Expressions

Modeling variables are typically used in expressions that define constraints or objective functions. Expressions are represented by objects of type `IloNumExpr`. They are built using methods such as `sum`, `prod`, `diff`, `negative`, and `square`. For example, the expression

```
x1 + 2*x2
```

where `x1` and `x2` are `IloNumVar` objects, is constructed by calling:

```
IloNumExpr expr = cplex.sum(x1, cplex.prod(2.0, x2));
```

It follows that a single variable is a special case of an expression, since `IloNumVar` is an extension of `IloNumExpr`.

The special case of linear expressions is represented by objects of type `IloLinearNumExpr`. Such expressions are editable, which is especially convenient when building up linear expressions in a loop, like this:

```
IloLinearNumExpr lin = cplex.linearNumExpr();
 for (int i = 0; i < num; ++i)
   lin.addTerm(value[i], variable[i]);
```

It should be noted that the special case of the scalar product of an array of values with an array of variables is directly supported through the method `scalProd`. Thus the above loop can be rewritten as:

```
IloLinearNumExpr lin = cplex.scalProd(value, variable);
```

It is recommended that you build expressions in terms of data that is either integer or double-precision (64 bit) floating-point. Single-precision (32 bit) floating-point data should be avoided as it can result in unnecessarily ill-conditioned problems. For more information, refer to *Numerical Difficulties* on page 196.

Ranged Constraints

Ranged constraints are constraints of the form: `lb` ≤ `expression` ≤ `ub` and are represented in Concert Technology by objects of type `IloRange`. The most general constructor is:

```
IloRange rng = cplex.range(lb, expr, ub, name);
```

where `lb` and `ub` are double values, `expr` is of type `IloNumExpr`, and `name` is a string.

By choosing the range bounds appropriately, ranged constraints can be used to model any of the more commonly found constraints of the form:

```
expr  relation  rhs,
```

where *`relation`* is the relation =, ≤, or ≥. The following table shows how to choose `lb` and `ub` for modeling these relations:

relation	lb	ub	method
=	`rhs`	`rhs`	`eq`
≤	`-Double.MAX_VALUE`	`rhs`	`le`
≥	`rhs`	`Double.MAX_VALUE`	`ge`

The last column contains the method provided with `IloModeler` to use directly to create the appropriate ranged constraint, when you specify the expression and right-hand side (RHS). For example, the constraint `expr` ≤ `1.0` is created by calling

```
IloRange le = cplex.le(expr, 1.0);
```

Again, all constructors for ranged constraints come in pairs, one constructor with and one without a name parameter.

Objective Functions

The objective function in Concert Technology is represented by objects of type `IloObjective`. Such objects are defined by an optimization sense, an expression, and an optional name. The objective expression is represented by an `IloNumExpr`. The objective sense is represented by an object of class `IloObjectiveSense` and can take two values, `IloObjectiveSense.Maximize` or `IloObjectiveSense.Minimize`. The most general constructor for an objective function object is:

```
IloObjective obj = cplex.objective(sense, expr, name);
```

where `sense` is of type `IloObjectiveSense`, `expr` is of type `IloNumExpr`, and `name` is a string.

For convenience, the methods `maximize` and `minimize` are provided to create a maximization or minimization objective respectively, without using an `IloObjectiveSense` parameter. Names for objective function objects are optional, so all constructor methods come in pairs, one with and one without the name parameter.

The Active Model

Modeling objects, constraints and objective functions, created as explained in *Modeling with IloModeler* on page 76, are now added to the active model. The active model is the model implemented by the `IloCplex` object itself. In fact, `IloModeler` is an extension of the `IloModel` interface defining the model API. Thus, `IloCplex` implements `IloModel`, or in other words, an `IloCplex` object *is* a model. The model implemented by the `IloCplex` object itself is referred to as the active model of the `IloCplex` object, or if there is no possibility of confusion between several optimizers, simply as the active model.

A model is just a set of modeling objects of type `IloAddable` such as `IloObjective` and `IloRange`. Objects of classes implementing this interface can be added to an instance of `IloModel`. Other `IloAddable` objects usable with `IloCplex` are `IloLPMatrix`, `IloConversion`, `IloSOS1`, and `IloSOS2`. These will be covered in the `IloMPModeler` section.

Variables cannot be added to a model because `IloNumVar` is not an extension of `IloAddable`. All variables used by other modeling objects (`IloAddable` objects) that have been added to a model are implicitly part of this optimization model. The explicit addition of a variable to a model can thus be avoided.

During modeling, a typical sequence of operations is to create a modeling object and immediately add it to the active model. To facilitate this, for most constructors with a name such as *`ConstructorName`*, there is also a method `add`*`ConstructorName`* which immediately adds the newly constructed modeling object to the active model. For example, the call

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

is equivalent to

```
IloObjective obj = cplex.add(cplex.maximize(expr));
```

Not only do the `add`*`ConstrucorName`* methods simplify the program, they are also more efficient than the two equivalent calls because an intermediate copy can be avoided.

Building the Model

All the building blocks are now in place to implement a method that creates a model. The diet problem consists of finding the least expensive diet using a set of foods such that all nutritional requirements are satisfied. The example in this chapter builds the specific diet model, chooses an optimizing algorithm, and shows how to access more detailed information about the solution.

The example includes a set of foods, where food `j` has a unit cost of `foodCost[j]`. The minimum and maximum amount of food `j` which can be used in the diet is designated

foodMin[j] and foodMax[j], respectively. Each food j also has a nutritional value nutrPerFood[i][j] for all possible nutrients i. The nutritional requirement states that in the diet the amount of every nutrient i consumed must be within the bounds nutrMin[i] and nutrMax[i].

Mathematically, this problem can be modeled using a variable Buy[j] for each food j indicating the amount of food j to buy for the diet. Then the objective is:

minimize $\sum_j$ (Buy[j] * foodCost[j])

The nutritional requirements mean that the following conditions must be observed; that is, for all i:

nutriMin[i] $\leq \sum_i$ nutrPerFood[i][j] * Buy[j] $\leq$ nutriMax[i]

Finally, every food must be within its bounds; that is, for all j:

foodMin[j] $\leq$ Buy[j] $\leq$ foodMax[j]

With what you have learned so far, you can implement a method that creates such a model.

```
static void buildModelByRow(IloModeler    model,
                            Data          data,
                            IloNumVar[]   Buy,
                            IloNumVarType type)
                                             throws IloException {
  int nFoods = data.nFoods;
  int nNutrs = data.nNutrs;

  for (int j = 0; j < nFoods; j++) {
    Buy[j] = model.numVar(data.foodMin[j], data.foodMax[j], type);
  }
  model.addMinimize(model.scalProd(data.foodCost, Buy));

  for (int i = 0; i < nNutrs; i++) {
    model.addRange(data.nutrMin[i],
                   model.scalProd(data.nutrPerFood[i], Buy),
                   data.nutrMax[i]);
  }
}
```

The function receives several parameters. The parameter model is used for two things:

- creating other modeling objects, and
- representing the model being created.

The argument data contains the data for the model to be built. The argument Buy is an array, initialized to length data.nFoods, containing the model's variables. Finally, parameter type is used to specify the type of the variables being created.

The function starts by creating the modeling variables, one by one, and storing them in array Buy. Each variable j is initialized to have bounds data.foodMin[j] and data.foodMax[j] and to be of type type.

The variables are first used to construct the objective function expression with method `model.scalProd(foodCost, Buy)`. This expression is immediately used to create the minimization objective which is directly added to the active model by `addMinimize`.

In the loop that follows, the nutritional constraints are added. For each nutrient `i` the expression representing the amount of nutrient in a diet with food levels `Buy` is computed using `model.scalProd(nutrPerFood[i], Buy)`. This amount of nutrient must be within the ranged constraint bounds `nutrMin[i]` and `nutrMax[i]`. This constraint is created and added to the active model with `addRange`.

Note that function `buildModelByRow` uses interface `IloModeler` rather than `IloCplex`. This allows the function to be called without change in another implementation of `IloModeler`, such as `IloSolver`.

Solving the Model

Once you have created an optimization problem in your active model, the `IloCplex` object is ready to solve it. This is done, for a model represented by `cplex` by calling:

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

The solve method returns a Boolean indicating whether or not a feasible solution was found and can be queried. However, when `true` is returned, the solution that was found may not be the optimal one; for example the optimization may have terminated prematurely because it ran into an iteration limit.

Additional information about a possible solution available in the `IloCplex` object can be queried with the method `getStatus` returning an `IloCplex.Status` object. Possible statuses are summarized in Table 2.2.

Table 2.2 *Solution Status*

Return Status	Active Model
`Error`	It has not been possible to process the active model, or an error occurred during the optimization.
`Unknown`	It has not been possible to process the active model far enough to prove anything about it. A common reason may be that a time limit was reached.
`Feasible`	A feasible solution for the model has been proven to exist.
`Bounded`	It has been proven that the active model has a finite bound in the direction of optimization. However, this does not imply the existence of a feasible solution.

Table 2.2 *Solution Status (Continued)*

Return Status	Active Model
`Optimal`	The active model has been solved to optimality. The optimal solution can be queried.
`Infeasible`	The active model has been proven to possess no feasible solution.
`Unbounded`	The active model has been proven to be unbounded. The notion of unboundedness adopted by `IloCplex` is technically that of dual infeasibility; this does not include the notion that the model has been proven to be feasible. Instead, what has been proven is that if there is a feasible solution with objective value z*, there exists a feasible solution with objective value z*-1 for a minimization problem, or z*+1 for a maximization problem.
`InfeasibleOrUnbounded`	The active model has been proven to be infeasible or unbounded.

For example, an `Optimal` status indicates that an optimal solution has been found and can be queried, whereas an `Infeasible` status indicates that the active model has been proven to be infeasible. See the online *ILOG CPLEX Java Reference Manual* for more information about these statuses.

More detailed information about the status of the optimizer can be queried with method `getCplexStatus` returning an object corresponding to ILOG CPLEX status codes. Again the online *ILOG CPLEX Java Reference Manual* contains further information about this.

Accessing Solution Information

If a solution was found with the `solve` method, it can be accessed and then queried using a variety of methods. The objective function can be accessed by calling

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

The values of individual modeling variables for the solution are accessed by calling methods `IloCplex.getValue`, for example:

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

Frequently, solution values for an array of variables are needed. Rather than having to implement a loop to query the solution values variable by variable, the method `IloCplex.getValues` is provided to do so with only one function call:

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

Similarly, slack values can be queried for the constraints in the active model using the methods `IloCplex.getSlack` or `IloCplex.getSlacks`.

Printing the Solution to the Diet Model

This can now be applied to solving the diet problem discussed earlier, and printing its solution.

```
IloCplex     cplex = new IloCplex();
IloNumVar[]  Buy   = new IloNumVar[nFoods];

if ( byColumn ) buildModelByColumn(cplex, data, Buy, varType);
else            buildModelByRow   (cplex, data, Buy, varType);

// Solve model

if ( cplex.solve() ) {
  System.out.println();
  System.out.println("Solution status = " + cplex.getStatus());
  System.out.println();
  System.out.println(" cost = " + cplex.getObjValue());
  for (int i = 0; i < nFoods; i++) {
    System.out.println(" Buy" + i + " = " +
                       cplex.getValue(Buy[i]));
  }
  System.out.println();
}
```

These lines of code start by creating a new `IloCplex` object and passing it, along with the raw data in another object, either to the method `buildModelByColumn` or to the method `buildModelByRow`. The array of variables returned by it is saved as the array `Buy`. Then the method `solve` is called to optimize the active model and, upon success, solution information is printed.

Choosing an Optimizer

The algorithm used in the `solve` methods can be controlled and if necessary tailored to the particular needs of the model. The most important control is that of selecting the optimizer. For solving the active model, ILOG CPLEX solves one continuous relaxation or a series of continuous relaxations.

- A single LP is solved if `IloCplex.isMIP`, `IloCplex.isQO`, and `IloCplex.isQC` return `false`. This is the case if the active model does **not** include:
  - integer variables, Boolean variables, or semi-continuous variables;
  - special ordered sets;

- piecewise linear functions among the constraints; or
- quadratic terms in the objective function or among the constraints.

`IloCplex` provides several optimizing algorithms to solve LPs. For more about those optimizers, see Chapter 8, *Solving LPs: Simplex Optimizers*, Chapter 9, *Solving LPs: Barrier Optimizer*, and Chapter 10, *Solving Network-Flow Problems* in this manual.

◆ A single QP is solved if both `IloCplex.isMIP` and `IloCplex.isQC` return `false` and `IloCplex.isQO` returns `true`. This is the case if the active model contains a quadratic (and positive semi-definite) objective but does **not** contain:

- integer variables, Boolean variables, or semi-continuous variables;
- quadratic terms among the constraints;
- special ordered sets; or
- piecewise linear functions among the constraints.

As in the case of LPs, `IloCplex` provides several optimizing algorithms to solve QPs. For more about identifying this kind of problem, see Chapter 11, *Solving Problems with a Quadratic Objective (QP)*.

◆ A single QCP is solved if `IloCplex.isMIP` returns `false` and `IloCplex.isQC` returns `true`, indicating that it detected a quadratically constrained program (QCP). This is the case if the active model contains one or more quadratic (and positive semi-definite) constraints but does **not** contain:

- integer variables, Boolean variables, or semi-continuous variables;
- special ordered sets; or
- piecewise linear functions.

`IloCplex` solves QCP models using the barrier optimizer. For more about this kind of problem, see Chapter 12, *Solving Problems with Quadratic Constraints (QCP)*.

In short, an LP model has a linear objective function and linear constraints; a QP model has a quadratic objective function and linear constraints; a QCP includes quadratic constraints, and it may have a linear or quadratic objective function. A problem that that can be represented as LP, QP, or QCP is also known collectively as a *continuous model* or a *continuous relaxation*.

A *series of relaxations* is solved if the active model is a MIP, which can be recognized by `IloCplex.isMIP` returning `true`. This is the case if the model contains any of the objects excluded for single continuous models. If a MIP contains a purely linear objective function, (that is, `IloCplex.isQO` returns `false`), the problem is more precisely called an MILP. Otherwise it is called an MIQP or MIQCP. MIPs are solved using branch & cut search, explained in more detail in Chapter 13, *Solving Mixed Integer Programming Problems (MIP)*.

Solving a Single Continous Model

To choose the optimizer to solve a single continous model, or the first continuous relaxation in a series, use

```
IloCplex.setParam(IloCplex.IntParam.RootAlg, alg)
```

where `alg` is an integer specifying the algorithm type. Table 2.3 shows you the available types of algorithms.

Table 2.3 *Algorithm Types for RootAlg*

alg	Algorithm Type	LP?	QP?	QCP?
0	IloCplex.Algorithm.Auto	yes	yes	yes
1	IloCplex.Algorithm.Primal	yes	yes	not available
2	IloCplex.Algorithm.Dual	yes	yes	not available
3	IloCplex.Algorithm.Network	yes	yes	not available
4	IloCplex.Algorithm.Barrier	yes	yes	yes
5	IloCplex.Algorithm.Sifting	yes	not available	not available
6	IloCplex.Algorithm.Concurrent	yes	yes	not available

You are not obliged to set this parameter. In fact, if you do not explicitly call `IloCplex.setParam(IloCplex.IntParam.RootAlg, alg)`, ILOG CPLEX will use the default: `IloCplex.Algorithm.Auto`. Any invalid setting will produce an error message.

The `IloCplex.Algorithm.Sifting` algorithm is not available for QP. `IloCplex` will default to the `IloCplex.Algorithm.Auto` setting when the parameter `IloCplex.IntParam.RootAlg` is set to `IloCplex.Algorithm.Sifting` for a QP.

Only the settings `IloCplex.Algorithm.Auto and IloCplex.Algorithm.Barrier` are valid for a QCP.

Solving Subsequent Continuous Relaxations in a MIP

Parameter `IloCplex.IntParam.RootAlg` also controls the algorithm used for solving the first continuous relaxation when solving a MIP. The algorithm for solving all subsequent

continous relaxations is then controlled by the parameter `IloCplex.IntParam.NodeAlg`. The algorithm choices appear in Table 2.4

Table 2.4 *Algorithm Types for NodeAlg*

alg	Algorithm Type	MILP?	MIQP?	MIQCP?
0	IloCplex.Algorithm.Auto	yes	yes	yes
1	IloCplex.Algorithm.Primal	yes	yes	not available
2	IloCplex.Algorithm.Dual	yes	yes	not available
3	IloCplex.Algorithm.Network	yes	not available	not available
4	IloCplex.Algorithm.Barrier	yes	yes	yes
5	IloCplex.Algorithm.Sifting	yes	not available	not available

Controlling ILOG CPLEX Optimizers

Though ILOG CPLEX defaults will prove sufficient to solve most problems, ILOG CPLEX offers a variety of other parameters to control various algorithmic choices. ILOG CPLEX parameters can take values of type `boolean`, `int`, `double`, and `string`. The parameters are accessed via parameter names defined in classes `IloCplex.BooleanParam`, `IloCplex.IntParam`, `IloCplex.DoubleParam`, and `IloCplex.StringParam` corresponding to the parameter type.

Parameters

Parameters are manipulated by means of `IloCplex.setParam`. For example:

```
cplex.setParam(IloCplex.BooleanParam.PreInd, false);
```

sets the Boolean parameter `PreInd` to `false`, instructing ILOG CPLEX not to apply presolve before solving the problem.

Integer parameters often indicate a choice from a numbered list of possibilities, rather than a quantity. For example, the class `IloCplex.PrimalPricing` defines constants with the integer parameters shown in Table 2.5, *Constants in IloCplex.PrimalPricing* for better maintainability of the code.

Table 2.5 *Constants in IloCplex.PrimalPricing*

Integer Parameter	Constant in class `IloCplex.PrimalPricing`
0	`IloCplex.PrimalPricing.Auto`
1	`IloCplex.PrimalPricing.Devex`
2	`IloCplex.PrimalPricing.Steep`
3	`IloCplex.PrimalPricing.SteepQStart`
4	`IloCplex.PrimalPricing.Full`

Thus, the suggested method for setting steepest-edge pricing for use with the primal simplex algorithm looks like this:

```
cplex.setParam(IloCplex.IntParam.PPriInd,
               IloCplex.PrimalPricing.Steep);
```

Table 2.6 gives an overview of the classes defining constants for parameters.

Table 2.6 *Classes with Parameters Defined by Integers.*

class	for use with parameters:
`IloCplex.Algorithm`	`IloCplex.IntParam.RootAlg` `IloCplex.IntParam.NodeAlg`
`IloCplex.MIPEmphasis`	`IloCplex.IntParam.MIPEmphasis`
`IloCplex.VariableSelect`	`IloCplex.IntParam.VarSel`
`IloCplex.NodeSelect`	`IloCplex.IntParam.NodeSel`
`IloCplex.DualPricing`	`IloCplex.IntParam.DPriInd`
`IloCplex.PrimalPricing`	`IloCplex.IntParam.PPriInd`

Parameters can be queried with method `IloCplex.getParam` and reset to their default settings with method `IloCplex.setDefaults`. The minimum and maximum value to which an integer or double parameter can be set is queried with methods `IloCplex.getMin` and `IloCplex.getMax`, respectively. The default value of a parameter is obtained with `IloCplex.getDefault`.

Priority Orders and Branching Directions

When solving MIPs, another important way to control the solution process is by providing priority orders and branching directions for variables. The methods for doing so are:

- `IloCplex.setDirection`,
- `IloCplex.setDirections`,
- `IloCplex.setPriority`, and
- `IloCplex.setPriorities`.

Priority orders and branch directions allow you to control the branching performed during branch & cut in a static way.

Dynamic control of the solution process of MIPs is provided through goals or control callbacks. Goals are discussed for C++ in *Using Goals* on page 453. Control callbacks are discussed in *Using Callbacks* on page 477. (Java goals and callbacks are similar to the C++ versions.) Goals and callbacks allow you to control the solution process when solving MIPs based on information generated during the solution process itself. *Goals and Callbacks: a Comparison* on page 505 contrasts the advantages of both.

More Solution Information

Depending on the model being solved and the algorithm being used, more solution information is generated in addition to the objective value and solution values for variables and slacks. The following sections explain how to access that additional information.

- *Writing Solution Files* on page 88
- *Dual Solution Information* on page 89
- *Basis Information* on page 89
- *Writing Solution Files* on page 88
- *Infeasible Solution Information* on page 89
- *Solution Quality* on page 90

Writing Solution Files

The class `IloCplex` offers a variety of ways to write information about a solution that it has found.

If you have used the barrier optimizer without crossover, for example, you can call the method `IloCplex.writeVectors` to write solution information into a file in VEC format. That format is documented in the reference manual ILOG CPLEX File Formats. The barrier optimizer is explained in detail in *Solving LPs: Barrier Optimizer* on page 225.

After solving, you can call the method `IloCplex.writeMIPstart` to write a MIP basis suitable for a restart. The file it writes is in MST format. That format is documented in the reference manual *ILOG CPLEX File Formats*.

The method `IloCplex.exportModel` writes the active model to a file. The format of the file depends on the file extension in the name of the file that your application passes as an argument to this method. A model exported in this way to a file can be read back into ILOG CPLEX by means of the method `IloCplex.importModel`. Both these methods are documented more fully in the reference manual of the Java API.

Dual Solution Information

When solving an LP or QP, all the algorithms also compute dual solution information that your application can then query. (However, no dual information is available for QCP models.) You can access reduced costs by calling the method `IloCplex.getReducedCost` or `IloCplex.getReducedCosts`. Similarly, you can access dual solution values for the ranged constraints of the active model by using the methods `IloCplex.getDual` or `IloCplex.getDuals`.

Basis Information

When solving an LP using all but `IloCplex.Algorithm.Barrier` without crossover, or when solving a QP with a Simplex optimizer, basis information is available as well. Basis information can be queried for the variables and ranged constraints of the active model using method `IloCplex.getBasisStatus`. This method returns basis statuses for the variables or constraints using objects of type `IloCplex.BasisStatus`, with possible values:

`IloCplex.BasisStatus.Basic`,

`IloCplex.BasisStatus.AtLower`,

`IloCplex.BasisStatus.AtUpper, and`

`IloCplex.BasisStatus.FreeOrSuperbasic`.

The availability of a basis for an LP allows you to perform sensitivity analysis for your model. Such analysis tells you by how much you can modify your model without affecting the solution you found. The modifications supported by the sensitivity analysis function include variable bound changes, changes to the bounds of ranged constraints, and changes to the objective function. They are analyzed by methods `IloCplex.getBoundSA`, `IloCplex.getRangeSA`, `IloCplex.getRHSSA` and `IloCplex.getObjSA`, respectively.

Infeasible Solution Information

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

Unfortunately, there is no simple comprehensive rule about whether or not current solution information can be queried. This is because by default, ILOG CPLEX uses a presolve

procedure to simplify the model. If, for example, the model is proven to be infeasible during the presolve, no current solution is generated by the optimizer. If, in contrast, infeasibility is only proven by the optimizer, current solution information is available to be queried. The status returned by calling `cplex.getCplexStatus` may help to determine which case you are facing, but it is probably safer and easier to include the methods for querying the solution within `try / catch` statements.

The method `IloCplex.isPrimalFeasible` can be called to learn whether a primal feasible solution has been found and can be queried. Similarly, the method `IloCplex.isDualFeasible` can be called to learn whether a dual feasible solution has been found and can be queried.

When an LP has been proven to be infeasible, ILOG CPLEX provides assistance for determining the cause of the infeasibility. This is done by computing what is known as an irreducibly inconsistent set (IIS), which is a description of the minimal subproblem that is still infeasible. An infeasible subproblem is minimal if, when you remove any of the constraints (including finite bounds), the infeasibility vanishes.

An IIS is computed for an infeasible model by calling method `cplex.getIIS`. Currently this method requires that the infeasibility was proven by the primal simplex method. It returns an object of type `IloCplex.IIS`, which can then be queried to find out what variables and constraints of the infeasible active model take part in the IIS.

If your problem is infeasible, you might also want to consider the method `IloCplex.feasOpt` to explore whether there are modifications you can make that would render your model feasible. *More about Infeasibility: FeasOpt* on page 210 explains that feature of ILOG CPLEX more fully, with examples of its use.

Solution Quality

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

`IloCplex` provides the method `getQuality` to allow you to analyze the quality of the solution. Several quality measures are defined in class `IloCplex.QualityType`. For example, to query the maximal bound violation of variables or slacks of the solution found by `cplex.solve` call:

```
IloCplex.QualityType inf =
cplex.getQuality(IloCplex.QualityType.MaxPrimalInfeas);
   double maxinfeas = inf.getValue();
```

The variable or constraint for which this maximum infeasibility occurs can be queried by calling `inf.getNumVar` or `inf.getRange`, one of which returns `null`. Not all quality

measures are available for solutions generated by different optimizers. See the *ILOG CPLEX Java Reference Manual* for further details.

Advanced Modeling with IloLPMatrix

So far the constraints have been considered only individually as ranged constraints of type `IloRange`; this approach is known as *modeling by row*s. However, mathematically the models that can be solved with `IloCplex` are frequently represented as:

Minimize (or Maximize) f(x)

such that $L \leq Ax \leq U$

with these bounds $L \leq x \leq U$

where A is a *sparse matrix*. A sparse matrix is one in which a significant portion of the coefficients are zero, so algorithms and data structures can be designed to take advantage of it by storing and working with the substantially smaller subset of nonzero coefficients.

Objects of type `IloLPMatrix` are provided for use with `IloCplex` to express constraint matrices rather than individual constraints. An `IloLPMatrix` object allows you to view a set of ranged constraints and the variables used by them as a matrix, that is, as: $L \leq Ax \leq U$

Every row of an `IloLPMatrix` object corresponds to an `IloRange` constraint, and every column of an `IloLPMatrix` object corresponds to a modeling variable (an instance of `IloNumVar`).

An `IloLPMatrix` object is created with the method `LPMatrix` defined in `IloMPModeler` like this:

```
IloLPMatrix lp = cplex.LPMatrix();
```

(or `cplex.addLPMatrix` to add it immediately to the active model). The rows and columns are then added to it by specifying the non-zero matrix coefficients. Alternatively, you can add complete `IloRange` and `IloNumVar` objects to it to create new rows and columns. When adding ranged constraints, columns will be implicitly added for all the variables in the constraint expression that do not already correspond to a column of the `IloLPMatrix`. The `IloLPMatrix` object will make sure of consistency between the mapping of rows to constraints and columns to variables. For example, if a ranged constraint that uses variables not yet part of the `IloLPMatrix` is added to the `IloLPMatrix`, new columns will automatically be added and associated to those variables.

See the online *ILOG CPLEX Java Reference Manual* for more information about `IloLPMatrix` methods.

Modeling by Column

The concept of modeling by column modeling comes from the matrix view of mathematical programming problems. Starting from a (degenerate) constraint matrix with all its rows but no columns, you populate it by adding columns to it. The columns of the constraint matrix correspond to variables.

Modeling by column in ILOG CPLEX is not limited to `IloLPMatrix`, but can be approached through `IloObjective` and `IloRange` objects as well. In short, for ILOG CPLEX, modeling by column can be more generally understood as using columns to hold a place for new variables to install in modeling objects, such as an objective or row. The variables are created as explained in the procedure.

Procedure for Modeling by Column

Start by creating a description of how to install a new variable into existing modeling objects. Such a description is represented by `IloColumn` objects. Individual `IloColumn` objects define how to install a new variable in one existing modeling object and are created with one of the `IloMPModeler.column` methods. Several `IloColumn` objects can be linked together (with the `IloCplex.add` method) to install a new variable in all modeling objects in which it is to appear. For example:

```
IloColumn col = cplex.column(obj, 1.0).and(cplex.column(rng, 2.0));
```

can be used to create a new variable and install it in the objective function represented by `obj` with a linear coefficient of `1.0` and in the ranged constraint `rng` with a linear coefficient of `2.0`.

Once the proper column object has been constructed, it can be used to create a new variable by passing it as the first parameter to the variable constructor. The newly created variable will be immediately installed in existing modeling objects as defined by the `IloColumn` object that has been used. So the line,

```
IloNumVar var = cplex.numVar(col, 0.0, 1.0);
```

creates a new variable with bounds `0.0` and `1.0` and immediately installs it in the objective `obj` with linear coefficient `1.0` and in the ranged constraint `rng` with linear coefficient `2.0`.

All constructor methods for variables come in pairs, one with and one without a first `IloColumn` parameter. Methods for constructing arrays of variables are also provided for modeling by column. These methods take an `IloColumnArray` object as a parameter that defines how each individual new variable is to be installed in existing modeling objects.

Optimizing the Diet Problem in Java

The problem solved in this example is to minimize the cost of a diet that satisfies certain nutritional constraints. You might also want to compare this approach through the Java API of ILOG CPLEX with similar applications in other programming languages:

- *Example: Optimizing the Diet Problem in C++* on page 61
- *Example: Optimizing the Diet Problem in C#.NET* on page 111
- *Example: Optimizing the Diet Problem in the Callable Library* on page 129

This example was chosen because it is simple enough to be viewed from a row as well as from a column perspective. Both ways are shown in the example. In this example, either perspective can be viewed as natural. Only one approach will seem natural for many models, but there is no general way of determining which is more appropriate (rows or columns) in a particular case.

The example accepts a filename and two options `-c` and `-i` as command line arguments. Option `-i` allows you to create a MIP model where the quantities of foods to purchase must be integers. Option `-c` can be used to build the model by columns.

The example starts by evaluating the command line arguments and reading the input data file. The input data of the diet problem is read from a file using an object of the embedded class `Diet.Data`. Its constructor requires a file name as an argument. Using the class `InputDataReader`, it reads the data from that file. This class is distributed with the examples, but will not be considered here as it does not use ILOG CPLEX or Concert Technology in any special way.

Creating the Optimizer

Once the data has been read, the `IloCplex` modeler/optimizer is created.

```
IloCplex     cplex = new IloCplex();
IloNumVar[]  Buy   = new IloNumVar[nFoods];

if ( byColumn ) buildModelByColumn(cplex, data, Buy, varType);
else            buildModelByRow   (cplex, data, Buy, varType);
```

Array `IloNumVar[] Buy` is also created where the modeling variables will be stored by `buildModelByRow` or `buildModelByColumn`.

You have already seen a method very similar to `buildModelByRow`. This function is called when `byColumn` is false, which is the case when the example is executed without the `-c` command line option; otherwise, `buildModelByColumn` is called. Note that unlike `buildModelByRow`, this method requires `IloMPModeler` rather than `IloModeler` as parameter since modeling by column is not available with `IloModeler`.

First, the function creates an empty minimization objective and empty ranged constraints, and adds them to the active model.

```
IloObjective cost       = model.addMinimize();
IloRange[]   constraint = new IloRange[nNutrs];

for (int i = 0; i < nNutrs; i++) {
   constraint[i] = model.addRange(data.nutrMin[i], data.nutrMax[i]);
}
```

Empty means that they use a `0` expression. After that the variables are created one by one, and installed in the objective and constraints modeling by column. For each variable, a column object must be created. Start by creating a column object for the objective by calling:

```
IloColumn col = model.column(cost, data.foodCost[j]);
```

The column is then expanded to include the coefficients for all the constraints using `col.and` with the column objects that are created for each constraint, as in the following loop:

```
for (int i = 0; i < nNutrs; i++) {
  col = col.and(model.column(constraint[i], data.nutrPerFood[i][j]));
}
```

When the full column object has been constructed it is finally used to create and install the new variable using:

```
Buy[j] = model.numVar(col, data.foodMin[j], data.foodMax[j], type);
```

Once the model is created, solving it and querying the solution is straightforward. What remains to be pointed out is the exception handling. In case of an error, ILOG CPLEX will throw an exception of type `IloException` or one of its subclasses. Thus the entire ILOG CPLEX program is enclosed in `try/catch` statements. The `InputDataReader` can throw exceptions of type `java.io.IOException` or `InputDataReader.InputDataReaderException`.

Since none of these three possible exceptions is handled elsewhere, the `main` function ends by catching them and issuing appropriate error messages.

The call to the method `cplex.end` frees the license that ILOG CPLEX uses.

Example: Diet.java Source Code

Here is the entire source code listing for the example. It is available in the distributed files for ILOG CPLEX Java

```
// --------------------------------------------------------------------------
// File: examples/src/Diet.java
// Version 9.0
// --------------------------------------------------------------------------
//  Copyright (C) 2001-2003 by ILOG.
```

```
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ------------------------------------------------------------------------
import ilog.concert.*;
import ilog.cplex.*;


public class Diet {
   static class Data {
      int        nFoods;
      int        nNutrs;
      double[]   foodCost;
      double[]   foodMin;
      double[]   foodMax;
      double[]   nutrMin;
      double[]   nutrMax;
      double[][] nutrPerFood;

      Data(String filename) throws IloException, java.io.IOException,
                                   InputDataReader.InputDataReaderException {
         InputDataReader reader = new InputDataReader(filename);

         foodCost = reader.readDoubleArray();
         foodMin  = reader.readDoubleArray();
         foodMax  = reader.readDoubleArray();
         nutrMin  = reader.readDoubleArray();
         nutrMax  = reader.readDoubleArray();
         nutrPerFood = reader.readDoubleArrayArray();

         nFoods = foodMax.length;
         nNutrs = nutrMax.length;

         if ( nFoods != foodMin.length  ||
              nFoods != foodMax.length    )
            throw new IloException("inconsistent data in file " + filename);
         if ( nNutrs != nutrMin.length    ||
              nNutrs != nutrPerFood.length  )
            throw new IloException("inconsistent data in file " + filename);
         for (int i = 0; i < nNutrs; ++i) {
            if ( nutrPerFood[i].length != nFoods )
               throw new IloException("inconsistent data in file " + filename);
         }
      }
   }

   static void buildModelByRow(IloModeler    model,
                               Data          data,
                               IloNumVar[]   Buy,
                               IloNumVarType type) throws IloException {
      int nFoods = data.nFoods;
      int nNutrs = data.nNutrs;
```

```
   for (int j = 0; j < nFoods; j++) {
      Buy[j] = model.numVar(data.foodMin[j], data.foodMax[j], type);
   }
   model.addMinimize(model.scalProd(data.foodCost, Buy));

   for (int i = 0; i < nNutrs; i++) {
      model.addRange(data.nutrMin[i],
                     model.scalProd(data.nutrPerFood[i], Buy),
                     data.nutrMax[i]);
   }
}

static void buildModelByColumn(IloMPModeler  model,
                               Data          data,
                               IloNumVar[]   Buy,
                               IloNumVarType type) throws IloException {
   int nFoods = data.nFoods;
   int nNutrs = data.nNutrs;

   IloObjective cost       = model.addMinimize();
   IloRange[]   constraint = new IloRange[nNutrs];

   for (int i = 0; i < nNutrs; i++) {
      constraint[i] = model.addRange(data.nutrMin[i], data.nutrMax[i]);
   }

   for (int j = 0; j < nFoods; j++) {
      IloColumn col = model.column(cost, data.foodCost[j]);
      for (int i = 0; i < nNutrs; i++) {
         col = col.and(model.column(constraint[i], data.nutrPerFood[i][j]));
      }
      Buy[j] = model.numVar(col, data.foodMin[j], data.foodMax[j], type);
   }
}


public static void main(String[] args) {

   try {
      String          filename  = "../../../examples/data/diet.dat";
      boolean         byColumn  = false;
      IloNumVarType   varType   = IloNumVarType.Float;

      for (int i = 0; i < args.length; i++) {
         if ( args[i].charAt(0) == '-') {
            switch (args[i].charAt(1)) {
            case 'c':
               byColumn = true;
               break;
            case 'i':
               varType = IloNumVarType.Int;
               break;
```

```
            default:
               usage();
               return;
            }
         }
         else {
            filename = args[i];
            break;
         }
      }

      Data data = new Data(filename);
      int nFoods = data.nFoods;
      int nNutrs = data.nNutrs;

      // Build model
      IloCplex     cplex = new IloCplex();
      IloNumVar[]  Buy   = new IloNumVar[nFoods];

      if ( byColumn ) buildModelByColumn(cplex, data, Buy, varType);
      else            buildModelByRow   (cplex, data, Buy, varType);

      // Solve model

      if ( cplex.solve() ) {
         System.out.println();
         System.out.println("Solution status = " + cplex.getStatus());
         System.out.println();
         System.out.println(" cost = " + cplex.getObjValue());
         for (int i = 0; i < nFoods; i++) {
            System.out.println(" Buy" + i + " = " + cplex.getValue(Buy[i]));
         }
         System.out.println();
      }
      cplex.end();
   }
   catch (IloException ex) {
      System.out.println("Concert Error: " + ex);
   }
   catch (InputDataReader.InputDataReaderException ex) {
      System.out.println("Data Error: " + ex);
   }
   catch (java.io.IOException ex) {
      System.out.println("IO Error: " + ex);
   }
}

static void usage() {
   System.out.println(" ");
   System.out.println("usage: java Diet [options] <data file>");
   System.out.println("options: -c  build model by column");
   System.out.println("         -i  use integer variables");
   System.out.println(" ");
```

```
   }
}
```

Modifying the Model

An important feature of ILOG CPLEX is that you can modify a previously created model to consider different scenarios. Furthermore, depending on the optimization model and algorithm used, ILOG CPLEX will save as much information from a previous solution as possible when optimizing a modified model.

The most important modification method is `IloModel.add`, for adding modeling objects to the active model. Conversely, `IloModel.remove` can be used to remove a previously added modeling object from a model.

When adding a modeling object such as a ranged constraint to a model, all the variables used by that modeling object implicitly become part of the model as well. However, when removing a modeling object, no variables are implicitly removed from the model. Instead, variables can only be explicitly removed from a model by calling `IloModel.delete`. This will cause the specified variables to be deleted from the model, and thus from all modeling objects in the model that are using these variables. In other words, deleting variables from a model may implicitly modify other modeling objects in that model.

The API of specific modeling objects may provide modification methods. For example, variable bounds can be changed using methods `IloNumVar.setLB` and `IloNumVar.setUB`. Similarly the bounds of ranged constraints can be changed using `IloRange.setLB` and `IloRange.setUB`.

Because not all the optimizers that implement the `IloModeler` interface support the ability to modify a model, modification methods are implemented in `IloMPModeler`. These methods are for manipulating the linear expressions in ranged constraints and objective functions used with `IloCplex`. The methods `IloMPModeler.setLinearCoef`, `IloMPModeler.setLinearCoefs`, and `IloMPModeler.addToExpr` apply in this situation.

The type of a variable cannot be changed. However, it can be overwritten for a particular model by adding an `IloConversion` object, which allows you to specify new types for variables within that model. When ILOG CPLEX finds a conversion object in the active model, it uses the variable types specified in the conversion object instead of the original type specified for the optimization. For example, in a model containing the following lines, ILOG CPLEX will only generate solutions where variable `x` is an integer (within tolerances), yet the type returned by `x.getType` will remain `IloNumVarType.Float`.

```
IloNumVar x = cplex.numVar(0.0, 1.0);
cplex.add(cplex.conversion(x, IloNumVarType.Int));
```

A variable can only be used in at most one conversion object, or the model will no longer be unambiguously defined. This does not imply that the type of a variable can only be changed

once and never again after that. Instead, you can remove the conversion object and add a new one to implement consecutive variable type changes.

C H A P T E R

3

ILOG Concert Technology for C#.NET Users

This chapter explores the features that ILOG CPLEX offers to users of C#.NET through Concert Technology. It walks you through an application based on the widely published diet problem. It includes these topics:

- *Describe* on page 102 contains the problem description.
- *Model* on page 104 shows how to represent the problem.
- *Solve* on page 108 demonstrates how to solve the problem and display the solution.
- *Good Programming Practices* on page 109 adds other features of the working application.
- *Example: Optimizing the Diet Problem in C#.NET* on page 111 tells you where to find the complete application and problem data.

The .NET API can be used from any progaming language in the .NET framework. This chapter concentrates on an example using C#.NET. There are also examples of VB.NET (Visual Basic in the .NET framework) delivered with ILOG CPLEX in *yourCPLEXhome*`\examples\i86_2000_7.1\vb`. Because of their .NET framework, those VB.NET examples differ from the traditional Visual Basic examples that may already be familiar to some ILOG CPLEX users. The traditional Visual Basic examples are available in *yourCPLEXhome*`\examples\msvc6\vb`.

> ***Note:*** *This chapter consists of a tutorial 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 tutorials. 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 ILOG Concert Technology for .NET users and ILOG CPLEX already installed and available in your integrated development environment.*

For hints about checking your installation of ILOG CPLEX and ILOG Concert Technology for .NET users, see the online manual *Getting Started*. It is also a good idea to try the tutorial for .NET users in that manual before beginning this one.

Describe

The aim of this tutorial is build a simple application with ILOG CPLEX and Concert Technology for .NET users. The tutorial is based on the well known diet problem: to minimize the cost of a daily diet that satisfies certain nutritional constraints. The conventional statement of the problem assumes data indicating the cost and nutritional value of each available food.

The finished application accepts a filename and two options `-c` and `-i` as command line arguments. Option `-i` allows you to create a MIP model where the quantities of foods to purchase must be integers (for example, 10 carrots). Otherwise, the application searches for a solution expressed in continuous variables (for example, 1.7 kilos of carrots). Option `-c` can be used to build the model by columns. Otherwise, the application builds the model by rows.

The finished application starts by evaluating the command line arguments and reading the input data file. The input data for this example is the same data as for the corresponding C++ and Java examples in this manual. The data is available in the standard distribution at:

yourCPLEXhome`\examples\data\diet.dat`

Step 1 Describe the Problem

Write a natural language description of the problem and answer these questions:

- What is known about this 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?

What is known?

The amount of nutrition provided by a given quantity of a given food.

The cost per unit of food.

The upper and lower bounds on the foods to be purchased for the diet

What are the unknowns?

The quantities of foods to buy.

What are the constraints?

The food bought to consume must satisfy basic nutritional requirements.

The amount of each food purchased must be within its bounds

What is the objective?

Minimize the cost of food to buy

Step 2 Open the file

Open the file *yourCPLEXhome*`\examples\src\tutorials\Dietlesson.cs` in your integrated development environment, such as Microsoft Visual Studio. Then go to the comment **Step 2** in `Dietlesson.cs`, and add the following lines to declare a class, a key element of this application.

```
public class Diet {
   internal class Data {
      internal int nFoods;
      internal int nNutrs;
      internal double[]   foodCost;
      internal double[]   foodMin;
      internal double[]   foodMax;
      internal double[]   nutrMin;
      internal double[]   nutrMax;
      internal double[][] nutrPerFood;

      internal Data(string filename) {
         InputDataReader reader = new InputDataReader(filename);

         foodCost = reader.ReadDoubleArray();
         foodMin  = reader.ReadDoubleArray();
         foodMax  = reader.ReadDoubleArray();
         nutrMin  = reader.ReadDoubleArray();
         nutrMax  = reader.ReadDoubleArray();
         nutrPerFood = reader.ReadDoubleArrayArray();
         nFoods = foodMax.Length;
         nNutrs = nutrMax.Length;

         if ( nFoods != foodMin.Length  ||
              nFoods != foodMax.Length    )
            throw new ILOG.CONCERT.Exception("inconsistent data in file "
                                             + filename);
         if ( nNutrs != nutrMin.Length    ||
              nNutrs != nutrPerFood.Length  )
            throw new ILOG.CONCERT.Exception("inconsistent data in file "
                                             + filename);
         for (int i = 0; i < nNutrs; ++i) {
            if ( nutrPerFood[i].Length != nFoods )
               throw new ILOG.CONCERT.Exception("inconsistent data in file "
                                             + filename);
         }
      }
   }
```

The input data of the diet problem is read from a file into an object of the nested class `Diet.Data`. Its constructor requires a file name as an argument. Using an object of the class `InputDataReader`, your application reads the data from that file.

Model

This example was chosen because it is simple enough to be viewed by rows as well as by columns. Both ways are implemented in the finished application. In this example, either perspective can be viewed as natural. Only one approach will seem natural for many models, but there is no general way of determining which is more appropriate (rows or columns) in a particular case.

Step 3 Create the model

Go to the comment **Step 3** in `Dietlesson.cs`, and add this statement to create the `Cplex` model for your application.

```
Cplex     cplex = new Cplex();
```

Step 4 Create an array to store the variables

Go to the comment **Step 4** in `Dietlesson.cs`, and add this statement to create the array of numeric variables that will appear in the solution.

```
INumVar[]  Buy   = new INumVar[nFoods];
```

At this point, only the array has been created, not the variables themselves. The variables will be created later as continuous or discrete, depending on user input. These numeric variables represent the unknowns: how much of each food to buy.

Step 5 Indicate by row or by column

Go to the comment **Step 5** in `Dietlesson.cs`, and add the following lines to indicate whether to build the problem by rows or by columns.

```
if ( byColumn ) BuildModelByColumn(cplex, data, Buy, varType);
else            BuildModelByRow   (cplex, data, Buy, varType);
```

The finished application interprets an option entered through the command line by the user to apply this conditional statement.

Build by Rows

The finished application is capable of building a model by rows or by columns, according to an option entered through the command line by the user. The next steps in this tutorial show you how to add a static method to your application. This method builds a model by rows.

Step 6 Set up rows

Go to the comment **Step 6** in `Dietlesson.cs`, and add the following lines to set up your application to build the model by rows.

```
internal static void BuildModelByRow(IModeler    model,
                                     Data        data,
                                     INumVar[]   Buy,
                                     NumVarType  type) {
   int nFoods = data.nFoods;
   int nNutrs = data.nNutrs;
```

Those lines begin the static method to build a model by rows. The next steps in this tutorial show you the heart of that method.

Step 7 Create the variables: build and populate by rows

Go to the comment **Step 7** in `Dietlesson.cs`, and add the following lines to create a loop that creates the variables of the problem with the bounds specified by the input data.

```
for (int j = 0; j < nFoods; j++) {
   Buy[j] = model.NumVar(data.foodMin[j], data.foodMax[j], type);
}
```

Step 8 Add objective

Go to the comment **Step 8** in `Dietlesson.cs`, and add this statement to add the objective to the model.

```
model.AddMinimize(model.ScalProd(data.foodCost, Buy));
```

The objective function indicates that you want to minimize the cost of the diet computed as the sum of the amount of each food to buy `Buy[i]` times the unit price of that food `data.foodCost[i]`.

Step 9 Add nutritional constraints

Go to the comment **Step 9** in `Dietlesson.cs`, and add the following lines to add the ranged nutritional constraints to the model.

```
      for (int i = 0; i < nNutrs; i++) {
         model.AddRange(data.nutrMin[i],
                        model.ScalProd(data.nutrPerFood[i], Buy),
                        data.nutrMax[i]);
      }
   }
```

Build by Columns

As noted in *Build by Rows* on page 105, the finished application is capable of building a model by rows or by columns, according to an option entered through the command line by the user. The next steps in this tutorial show you how to add a static method to your application to build a model by columns.

Step 10 Set up columns

Go to the comment **Step 10** in `Dietlesson.cs`, and add the following lines to set up your application to build the problem by columns.

```
   internal static void BuildModelByColumn(IMPModeler model,
                                           Data       data,
                                           INumVar[]  Buy,
                                           NumVarType type) {
      int nFoods = data.nFoods;
      int nNutrs = data.nNutrs;
```

Those lines begin a static method that the next steps will complete.

Step 11 Add empty objective function and constraints

Go to the comment **Step 11** in `Dietlesson.cs`, and add the following lines to create empty columns that will hold the objective and ranged constraints of your problem.

```
      IObjective cost       = model.AddMinimize();
      IRange[]   constraint = new IRange[nNutrs];

      for (int i = 0; i < nNutrs; i++) {
         constraint[i] = model.AddRange(data.nutrMin[i], data.nutrMax[i]);
      }
```

Step 12 Create variables

Go to the comment **Step 12** in `Dietlesson.cs`, and add the following lines to create each of the variables.

```
      for (int j = 0; j < nFoods; j++) {

         Column col = model.Column(cost, data.foodCost[j]);

         for (int i = 0; i < nNutrs; i++) {
            col = col.And(model.Column(constraint[i],
                                       data.nutrPerFood[i][j]));
         }

         Buy[j] = model.NumVar(col, data.foodMin[j], data.foodMax[j], type);

      }
   }
```

For each food `j`, a column object `col` is first created to represent how the new variable for that food is to be added to the objective function and constraints. Then that column object is used to construct the variable `Buy[j]` that represents the amount of food `j` to be purchased for the diet. At this time, the new variable will be installed in the objective function and constraints as defined by the column object `col`.

Solve

After you have added lines to your application to build a model, you are ready for the next steps: adding lines for solving and displaying the solution.

Step 13 Solve

Go to the comment **Step 13** in `Dietlesson.cs`, and add this statement to solve the problem.

```
         if ( cplex.Solve() ) {
```

Step 14 Display the solution

Go to the comment **Step 14** in `Dietlesson.cs`, and add the following lines to display the solution.

```
      System.Console.WriteLine();
      System.Console.WriteLine("Solution status = "
                               + cplex.GetStatus());
      System.Console.WriteLine();
      System.Console.WriteLine(" cost = " + cplex.ObjValue);
      for (int i = 0; i < nFoods; i++) {
         System.Console.WriteLine(" Buy"
                                  + i
                                  + " = "
                                  + cplex.GetValue(Buy[i]));
      }
      System.Console.WriteLine();
   }
```

Step 15 End and free license

Go to the comment **Step 15** in `Dietlesson.cs`, and add this statement to free the license used by ILOG CPLEX.

```
cplex.End();
```

Good Programming Practices

The next steps of this tutorial show you how to add features to your application.

Step 16 Read the command line (data from user)

Go to the comment **Step 16** in `Dietlesson.cs`, and add the following lines to read the data entered by the user at the command line.

```
for (int i = 0; i < args.Length; i++) {
   if ( args[i].ToCharArray()[0] == '-') {
      switch (args[i].ToCharArray()[1]) {
      case 'c':
         byColumn = true;
         break;
      case 'i':
         varType = NumVarType.Int;
         break;
      default:
         Usage();
         return;
      }
   }
   else {
      filename = args[i];
      break;
   }
}

Data data = new Data(filename);
```

Step 17 Show correct use of command line

Go to the comment **Step 17** in `Dietlesson.cs`, and add the following lines to show the user how to use the command correctly (in case of inappropriate input from a user).

```
internal static void Usage() {
   System.Console.WriteLine(" ");
   System.Console.WriteLine("usage: Diet [options] <data file>");
   System.Console.WriteLine("options: -c  build model by column");
   System.Console.WriteLine("         -i  use integer variables");
   System.Console.WriteLine(" ");
}
```

Step 18 Enclose the application in try catch statements

Go to the comment **Step 18** in `Dietlesson.cs`, and add the following lines to enclose your application in a try and catch statement in case of anomalies during execution.

```
      }
      catch (ILOG.CONCERT.Exception ex) {
         System.Console.WriteLine("Concert Error: " + ex);
      }
      catch (InputDataReader.InputDataReaderException ex) {
         System.Console.WriteLine("Data Error: " + ex);
      }
      catch (System.IO.IOException ex) {
         System.Console.WriteLine("IO Error: " + ex);
      }
   }
```

The try part of that try and catch statement is already available in your original copy of `Dietlesson.cs`. When you finish the steps of this tutorial, you will have a complete application ready to compile and execute.

Example: Optimizing the Diet Problem in C#.NET

You can see the complete program online at:

yourCPLEXhome`\examples\src\Diet.cs`

There is a project for this example, suitable for use in an integrated development environment, such as Microsoft Visual Studio, at:

yourCPLEXhome`\examples\i86_2000_7.1\`*format*`\Diet.csproj`

The empty lesson, suitable for interactively following this tutorial, is available at:

yourCPLEXhome`\examples\tutorials\Dietlesson.cs`

C H A P T E R

4

The ILOG CPLEX Callable Library

This chapter shows how to write C applications using the ILOG CPLEX Callable Library. It includes sections about:

- *Architecture of the ILOG CPLEX Callable Library* on page 113, including information about licensing and about compiling and linking your programs
- *Using the Callable Library in an Application* on page 115
- *ILOG CPLEX Programming Practices* on page 118
- *Managing Parameters from the Callable Library* on page 127
- *Example: Optimizing the Diet Problem in the Callable Library* on page 129

Architecture of the ILOG CPLEX Callable Library

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

You can use the Callable Library to write applications that conform to many modern computer programming paradigms, such as client-server applications within distributed environments, multithreaded applications running on multiple processors, applications

linked to database managers, or applications using flexible graphic user interface builders, just to name a few.

The Callable Library together with the ILOG CPLEX database make up the ILOG CPLEX *core*, as you see in Figure 4.1. The ILOG CPLEX database includes the computing environment, its communication channels, and your problem objects. You will associate the core with your application by calling library routines.

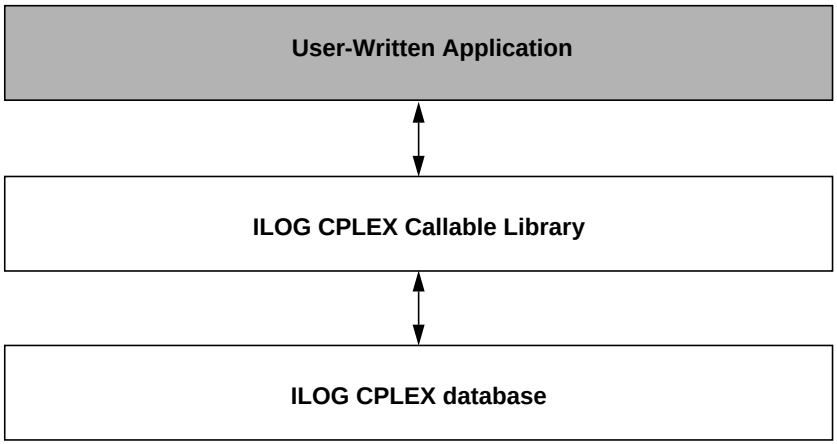

Figure 4.1 *A View of the ILOG CPLEX Callable Library*

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

- *problem modification routines* let you define a problem and change it after you have created it within the ILOG CPLEX database;
- *optimization routines* enable you to optimize a problem and generate results;
- *utility routines* handle application programming issues;
- *problem query routines* access information about a problem after you have created it;
- *file reading and writing routines* move information from the file system of your operating system into your application, or from your application into the file system;
- *parameter routines* enable you to query, set, or modify parameter values maintained by ILOG CPLEX.

Licenses

ILOG CPLEX runs under the control of the ILOG License Manager (ILM). Before you can run any application program that calls ILOG CPLEX, you must have established a valid

license that it can read. Licensing instructions are provided to you separately when you buy or upgrade ILOG CPLEX. Contact your local ILOG support department if this information has not been communicated to you or if you find that you need help in establishing your ILOG CPLEX license.

Compiling and Linking

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

Using the Callable Library in an Application

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

Initialize the ILOG CPLEX Environment

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

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

This routine checks for a valid ILOG CPLEX license and then returns a C pointer to the ILOG CPLEX environment that is creates. Your application then passes this C pointer to other ILOG CPLEX routines (except `CPXmsg`). As a developer, you decide for yourself whether the variable containing this pointer should be global or local in your application. Because the operation of checking the license can be relatively time consuming, it is strongly recommended that you call the `CPXopenCPLEX` routine only once, or as infrequently as possible, in a program that solves a sequence of problems.

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

Instantiate the Problem Object

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

Most applications will use only one problem object, though ILOG CPLEX allows you to create multiple problem objects within a given ILOG CPLEX environment.

Put Data in the Problem Object

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

- You can make a sequence of calls, in any convenient order, to these routines:
  - `CPXaddcolsex`
  - `CPXaddqconstr`
  - `CPXaddrows`
  - `CPXchgcoeflist`
  - CPXcopyctype
  - CPXcopyqsep
  - CPXcopyquad
  - `CPXnewcols`
  - `CPXnewrows`
- If data already exist in MPS, SAV, or LP format in a file, you can call `CPXreadcopyprob` to read that file and copy the data into the problem object. Mathematical Programming System (MPS) is an industry-standard format for organizing data in mathematical programming problems. LP and SAV file formats are ILOG CPLEX-specific formats for expressing linear programming problems as equations or inequalities. *Understanding File Formats* on page 156 explains these formats briefly. They are documented in the reference manual *ILOG CPLEX File Formats*.
- You can assemble arrays of data and then call `CPXcopylp` to copy the data into the problem object.

Whenever possible, compute your problem data in double precision (64 bit). Computers are finite-precision machines, and truncating your data to single precision (32 bit) can result in

unnecessarily ill-conditioned problems For more information, refer to *Numerical Difficulties* on page 196.

Optimize the Problem

Call one of the ILOG CPLEX optimizers to solve the problem object that you have instantiated and populated. *Choosing an Optimizer for Your LP Problem* on page 184 explains in greater detail how to choose an appropriate optimizer for your problem.

Change the Problem Object

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

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

Destroy the Problem Object

Use the routine `CPXfreeprob` to destroy a problem object when your application no longer needs it. Doing so will free all memory required to solve that problem instance.

Release the ILOG CPLEX Environment

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

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

ILOG CPLEX Programming Practices

This section lists some of the programming practices ILOG observes in developing and maintaining the ILOG CPLEX Callable Library.

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

Variable Names and Calling Conventions

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

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

```
int CPXgetobjval (CPXCENVptr env, CPXCLPptr lp, double *objval_p);
```

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

```
double objval;
```

Then call `CPXgetobjval` in this way:

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

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

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

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

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

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

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

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

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

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

For guidance about how to pass values to ILOG CPLEX routines from application languages such as FORTRAN or BASIC that conventionally call by reference, see *Call by Reference* on page 127 in this manual, and consult the documentation for those languages.

Data Types

In the Callable Library, ILOG CPLEX defines a few special data types for specific ILOG CPLEX objects, as you see in Table 4.1. The types starting with `CPXC` represent the corresponding pointers to constant (`const`) objects.

Table 4.1 *Special Data Types in the ILOG CPLEX Callable Library*

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

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

Ownership of Problem Data

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

Problem Size and Memory Allocation Issues

As indicated in *Change the Problem Object* on page 117, after you have created a problem object by calling `CPXcreateprob`, you can modify the problem in various ways through calls to routines from the Callable Library. There is no need for you to allocate extra space in anticipation of future problem modifications. Any limit on problem size is determined by system resources and the underlying implementation of the system function `malloc`—not by artificial limits in ILOG CPLEX.

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

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

Table 4.2 *Default Settings of ILOG CPLEX Growth Parameters*

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

Table 4.2 shows you the default settings of these growth parameters. At these defaults, if an application populates the problem object one row at a time, then ILOG CPLEX will cache the row additions until an updated problem is needed, for example when a query or optimization function is called. Similarly, it will cache column-based additions after 100 columns, and nonzero-based arrays when the additions of coefficients produces another 500

nonzeros to add to the matrix. *Memory Management and Problem Growth* on page 194 offers guidelines about performance tuning with these parameters.

Status and Return Values

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

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

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

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

It is extremely important that your application check the status—whether the status is indicated by the return value or by a parameter—of the routine that it calls before it proceeds.

Symbolic Constants

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

Parameter Routines

You can set many parameters in the ILOG CPLEX environment to control ILOG CPLEX operation. The values of these parameters may be integer, double, or character strings, so

there are sets of routines for accessing and setting them. Table 4.3 shows you the names and

Table 4.3 *Callable Library Routines for Parameters in the ILOG CPLEX Environment*

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

purpose of these routines. Each of these routines accepts the same first argument: a pointer to the ILOG CPLEX environment (that is, the pointer returned by CPXopenCPLEX). The second argument of each of those parameter routines is the parameter number, a symbolic constant defined in the header file, cplex.h. *Managing Parameters from the Callable Library* on page 127 offers more details about parameter settings.

Null Arguments

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

Row and Column References

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

Within the linear programming data structure, the rows and columns that represent constraints and variables are referenced by an *index number*. Each row and column may optionally have an associated name. If you add or delete rows, the index numbers usually change:

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

However, ILOG CPLEX updates the names so that each row or column index will correspond to the correct row or column name. Double checking names against index numbers is the only sure way to determine which changes may have been made to matrix

indices in such a context. The routines `CPXgetrowindex` and `CPXgetcolindex` translate names to indices.

Character Strings

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

Checking Problem Data

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

Using the Data Checking Parameter

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

- invalid `sense`/`ctype`/`sostype` values
- indices out of range, for example, `rowind` ≥ `numrows`
- duplicate entries
- `matbeg` or `sosbeg` array with decreasing values
- `NAN`s in double arrays
- `NULL`s in name arrays

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

Using Diagnostic Routines for Debugging

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

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

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

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

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

Callbacks

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

- during the presolve process;
- once per iteration in a linear programming or quadratic programming routine; and
- at various points, such as before each node is processed, in a mixed integer optimization.

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

Using Callbacks on page 477 explores callback facilities in greater detail.

Portability

ILOG CPLEX contains a number of features to help you create Callable Library applications that can be easily ported between UNIX and Windows platforms.

CPXPUBLIC

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

Function Pointers

All ILOG CPLEX Callable Library routines that require pointers to functions expect the passed-in pointers to be declared as `CPXPUBLIC`. Consequently, when your application uses such routines as `CPXaddfuncdest`, `CPXsetlpcallbackfunc`, and `CPXsetmipcallbackfunc`, it must declare the user-written callback functions with the `CPXPUBLIC` designation. For UNIX systems, this has no effect. For Win32 systems, this will cause the callback functions to be declared with the `__stdcall` calling convention. For examples of function pointers and callbacks, see *Example: Using Callbacks lpex4.c* on page 488 and *Example: Callable Library Message Channels* on page 164.

CPXCHARptr, CPXCCHARptr, and CPXVOIDptr

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

File Pointers

File pointer arguments for Callable Library routines should be declared with the type `CPXFILEptr`. On UNIX platforms, this practice is equivalent to using the file pointer type. On Win32 platforms, the file pointers declared this way will correspond to the environment of the ILOG CPLEX DLL. Any file pointer passed to a Callable Library routine should be obtained with a call to `CPXfopen` and closed with `CPXfclose`. Callable Library routines with file pointer arguments include `CPXsetlogfile`, `CPXaddfpdest`, `CPXdelfpdest`,

and CPXfputs. *Callable Library Routines for Message Channels* on page 163 discusses most of those routines.

String Functions

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

FORTRAN Interface

The Callable Library can be interfaced with FORTRAN applications. Although they are no longer distributed with the prodcut, you can download examples of a FORTRAN application from the ILOG web site. Direct your browser to this FTP site:

```
ftp://ftp.cplex.com/pub/examples
```

Those examples were compiled with CPLEX versions 7.0 and earlier on a particular platform. Since C-to-FORTRAN interfaces vary across platforms (operating system, hardware, compilers, etc.), you may need to modify the examples for your own system.

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

Case-Sensitivity

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

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

Underscore

On some systems, all FORTRAN external symbols are created with an underscore character (that is, _) added to the end of the symbol name. Some systems have an option to turn off this feature. If you are able to turn off those postpended underscores, you may not need other "glue" routines.

Six-Character Identifiers

FORTRAN 77 allows identifiers that are unique only up to six characters. However, in practice, most FORTRAN compilers allow you to exceed this limit. Since routines in the

Callable Library have names greater than six characters, you need to verify whether your FORTRAN compiler enforces this limit or allows longer identifiers.

Call by Reference

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

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

Pointers

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

Strings

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

C++ Interface

The ILOG CPLEX header file, `cplex.h`, includes the `extern` C statements necessary for use with C++. If you wish to call the ILOG CPLEX C interface from a C++ application, rather than using Concert Technology, you can include `cplex.h` in your C++ source.

Managing Parameters from the Callable Library

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

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

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

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

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

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

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

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

- the value you want the parameter to assume.

◆ `CPXsetstrparam` accepts arguments to indicate:

- the environment; that is, a pointer of type `CPXENVptr` returned by `CPXopenCPLEX`;
- the parameter to set; this routine sets parameters of type const `char*`;
- the value you want the parameter to assume.

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

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

Example: Optimizing the Diet Problem in the Callable Library

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

◆ *Creating a Model Row by Row* on page 130;

◆ *Creating a Model Column by Column* on page 130;

◆ *Solving the Model with CPXlpopt* on page 131.

Problem Representation

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

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

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

The diet problem is equally suited for both kinds of modeling. In fact you can even mix both approaches in the same program: If a new food product is introduced, you can create a new

variable for it, regardless of how the model was originally built. Similarly, is a new nutrient is discovered, you can add a new constraint for it.

Creating a Model Row by Row

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

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

$\texttt{nutrmin[i]} \leq \sum_j (\texttt{nutrper[i][j]} * \texttt{buy[j]}) \leq \texttt{nutrmax[i]}$

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

$\texttt{cost} = \sum_j (\texttt{cost[j]} * \texttt{buy[j]})$

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

Creating a Model Column by Column

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

$\texttt{nutrmin[i]} \leq \ldots \leq \texttt{nutrmax[i]}$

where `...` is left to be filled once you walk into your store. You also set up the objective function to minimize the cost. Constraint `i` is referred to as `rng[i]` and the objective is referred to as `cost`.

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

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

where the notation "`+`" and "`sum`" indicates that you “add” the new variable `j` to the objective `cost` and constraints `rng[i]`. The value in parentheses is the linear coefficient that is used for the new variable).

Program Description

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

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

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

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

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

Solving the Model with CPXlpopt

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

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

Complete Program

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

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/diet.c                                               */
/*  Version 9.0                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2003 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* diet.c - reading data for a dietary problem, building the model
   and solving it.  methods for creating the model:

      diet -r <datafile>   generates the problem by adding rows
      diet -c <datafile>   generates the problem by adding columns
 */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

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

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

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


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

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



int
main (int argc, char **argv)
```

```
{
   int status = 0;

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

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

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

   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           i, j;

   /* Check the command line arguments */

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

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


   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

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

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

```
/* Turn on output to the screen */

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

/* Turn on data checking */

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

/* Create the problem. */

lp = CPXcreateprob (env, &status, "diet");

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

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

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

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

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

/* Optimize the problem and obtain solution. */

status = CPXlpopt (env, lp);
if ( status ) {
```

```
      fprintf (stderr, "Failed to optimize LP.\n");
      goto TERMINATE;
   }

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

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

   /* Write the output to the screen. */

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

   for (j = 0; j < nfoods; j++) {
      printf ("Food %d:  Buy = %10f\n", j, x[j]);
   }

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

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

TERMINATE:

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

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

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

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

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

      if ( status > 0 ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
```

```
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

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

   return (status);

}  /* END main */


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

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

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

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

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

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

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

```
TERMINATE:

   free_and_null ((char **)&ind);

   return (status);

}  /* END populatebyrow */



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

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

   int i, j;

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

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

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

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

   ind = (int*) malloc(nfoods * sizeof(int));
   if ( ind == NULL ) {
      status = CPXERR_NO_MEMORY;
      goto TERMINATE;
```

```
   }
   for (i = 0; i < nnutr; i++) {
      ind[i] = i;
   }

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

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

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

TERMINATE:

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

   return (status);

}  /* END populatebycolumn */


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

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */



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


static int
readarray (FILE *in, int *num_p, double **data_p)
{
   int  status = 0;
```

```
   int  max, num;
   char ch;

   num = 0;
   max = 10;

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

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

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

   *num_p = num;

TERMINATE:

   return (status);

} /* END readarray */

static int
```

```
readdata (char* file,
          int *nfoods_p, double **cost_p, double **lb_p, double **ub_p,
          int *nnutr_p, double **nutrmin_p, double **nutrmax_p,
          double ***nutrper_p)
{
   int status = 0;

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

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

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

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

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

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

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

```
      }
      fscanf (in, "%c", &ch);
      if ( i < nmin-1  &&  ch != ',' ) {
         status = -1;
         goto TERMINATE;
      }
   }
   if ( ch != ']' ) {
      status = -1;
      goto TERMINATE;
   }
TERMINATE:
   return (status);
} /* END readdata */
```

Part II

Programming Considerations

This part of the manual documents concepts that are valid as you develop an application, regardless of the programming language that you choose. It highlights software engineering concepts implemented in ILOG CPLEX, concepts that will enable you to develop effective applications to exploit it efficiently. This part contains:

- *Developing CPLEX Applications* on page 145
- *Managing Input and Output* on page 155
- *Licensing an Application* on page 175

C H A P T E R

5

Developing CPLEX Applications

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

- *Tips for Successful Application Development* on page 145
- *Using the Interactive Optimizer for Debugging* on page 150
- *Eliminating Common Programming Errors* on page 151

Tips for Successful Application Development

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

Prototype the Model

Begin by creating a small-scale version of the model for your problem. (There are modeling languages, such as ILOG OPL, that may be helpful to you for this task.) This prototype

model can serve as a test-bed for your application and a point of reference during development.

Identify Routines to Use

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

Test Interactively

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

Most importantly, optimization commands in the Interactive Optimizer perform exactly like optimization routines in the Component Libraries. For an LP, the `optimize` command in the Interactive Optimizer works the same way as the `cplex.solve` and `CPXlpopt` routines in the ILOG CPLEX Component Libraries. Consequently, any discrepancy between the Interactive Optimizer and the Component Libraries routines with respect to the solutions found, memory used, or time taken indicates a problem in the logic of the application calling the routines.

Assemble Data Efficiently

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

- If your Callable Library application builds the arrays of the problem in memory and then calls `CPXcopylp`, it avoids time-consuming reads from disk files.
- In the Callable Library, using the routines `CPXnewcols`, `CPXnewrows`, `CPXaddcols`, `CPXaddrows`, and `CPXchgcoeflist` may help you build modular code that will be more easily modified and maintained than code that assembles all problem data in one step.

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

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

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

Test Data

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

Choose an Optimizer

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

- The primal simplex, dual simplex, and primal-dual barrier methods can be used to solve linear and quadratic programs.
- The barrier optimizer can also be used to solve quadratically constrained programming problems.
- The network optimizer is appropriate for solving linear and quadratic programs with large embedded networks.
- The branch & cut algorithm should be used if the problem contains discrete components (binary, integer, or semi-continuous variables, piecewise linear objective, or SOS sets).

In ILOG CPLEX, there are many possible parameter settings for each optimizer. Generally, the default parameter settings are best for linear programming and quadratic programming problems, but Chapter 8, *Solving LPs: Simplex Optimizers* and Chapter 11, *Solving Problems with a Quadratic Objective (QP)* offer more detail about improving performance with respect to these problems. Integer programming problems are more sensitive to specific

parameter settings, so you may need to experiment with them, as suggested in Chapter 13, *Solving Mixed Integer Programming Problems (MIP)*.

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

Program with a View toward Maintenance and Modifications

Good programming practices save development time and make an application easier to understand and modify. *Tips for Successful Application Development* on page 145 outlines ILOG programming conventions in developing ILOG CPLEX. In addition, the following programming practices are recommended.

Comment Your Code

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

Write Readable Code

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

Avoid Side-Effects

It is good idea to minimize side-effects by avoiding expressions that produce internal effects. In C, for example, try to avoid expressions of this form:

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

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

Don’t Change Argument Values

A user-defined function should not change the values of its arguments. Do not use an argument to a function on the left-hand side of an assignment statement in that function.

Since C and C++ pass arguments by value, treat the arguments strictly as values; do not change them inside a function.

Declare the Type of Return Values

Always declare the return type of functions explicitly. Though C has a "historical tradition" of making the default return type of all functions `int`, it is a good idea to declare explicitly the return type of functions that return a value, and to use `void` for procedures that do not return a value.

Manage the Flow of Your Code

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

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

Localize Variables

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

Name Your Constants

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

Choose Clarity First, Efficiency Later

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

Debug Effectively

Using Diagnostic Routines for Debugging on page 124, contains tips and guidelines for debugging an application that uses the ILOG CPLEX Callable Library. In that context, a symbolic debugger as well as other widely available development tools are quite helpful to produce error-free code.

Test Correctness, Test Performance

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

Using the Interactive Optimizer for Debugging

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

The first step in using the Interactive Optimizer for debugging is to write a version of the problem from the application into a formatted file that can then be loaded into the Interactive Optimizer. To do so, insert a call to the method `IloCplex::writeBasis` or to the routine `CPXwriteprob` into your application. Use that call to create a file, whether an LP, SAV, or MPS formatted problem file. (*Understanding File Formats* on page 156 briefly describes these file formats.) Then read that file into the Interactive Optimizer and optimize the problem there.

Note that MPS, LP and SAV files have differences that influence how to interpret the results of the Interactive Optimizer for debugging. SAV files contain the exact binary representation of the problem as it appears in your program, while MPS and LP files are text files containing possibly less precision for numeric data. And, unless every variable appears on the objective function, ILOG CPLEX will probably order the variables differently when it reads the problem from an LP file than from an MPS or SAV file. With this in mind, SAV files are the most useful for debugging using the Interactive Optimizer, followed by MPS files, then finally LP files, in terms of the change in behavior you might see by use of explicit

files. On the other hand, LP files are often quite helpful when you want to examine the problem, more so than as input for the Interactive Optimizer. Furthermore, try solving both the SAV and MPS files of the same problem using the Interactive Optimizer. Different results may provide additional insight into the source of the difficulty. In particular, use the following guidelines with respect to reproducing your program's behavior in the Interactive Optimizer.

1. If you can reproduce the behavior with a SAV file, but not with an MPS file, this suggests corruption or errors in the problem data arrays. Use the `DataCheck` parameter or diagnostic routines in the source file `check.c` to track down the problem.
2. If you can reproduce the behavior in neither the SAV file nor the MPS file, the most likely cause of the problem is that your program has some sort of memory error. Memory debugging tools such as Purify will usually find such problems quickly.
3. When solving a problem in MPS or LP format, if the Interactive Optimizer issues a message about a segmentation fault or similar ungraceful interruption and exits, contact ILOG CPLEX technical support to arrange for transferring the problem file. The Interactive Optimizer should never exit with a system interrupt when solving a problem from a text file, even if the program that created the file has errors. Such cases are extremely rare.

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

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

Eliminating Common Programming Errors

This section serves as a checklist to help you eliminate common pitfalls from your application. It includes the following topics:

◆ *Check Your Include Files* on page 152

- *Clean House and Try Again* on page 152
- *Read Your Messages* on page 152
- *Check Return Values* on page 152
- *Beware of Numbering Conventions* on page 153
- *Make Local Variables Temporarily Global* on page 153
- *Solve the Problem You Intended* on page 153
- *Special Considerations for Fortran* on page 153
- *Tell Us* on page 153

Check Your Include Files

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

Clean House and Try Again

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

Read Your Messages

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

To query exceptions in Concert Technology, use the methods:

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

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

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

Check Return Values

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

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

Beware of Numbering Conventions

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

Make Local Variables Temporarily Global

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

Solve the Problem You Intended

Your application may inadvertently alter the problem and thus produce unexpected results. To check whether your application is solving the problem you intended, use the Interactive Optimizer, as in *Using the Interactive Optimizer for Debugging* on page 150, and the diagnostic routines, as in *Using Diagnostic Routines for Debugging* on page 124.

You should not ignore any ILOG CPLEX warning message in this situation either, so read your messages, as in *Read Your Messages* on page 152.

If you are working in the Interactive Optimizer, you can use the command `display problem stats` to check the problem dimensions.

Special Considerations for Fortran

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

It is important that you use the Fortran declaration `IMPLICIT NONE` to help you detect any unintended type conversions, because such inadvertent conversions frequently lead to strange application behavior.

Tell Us

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

C H A P T E R

6

Managing Input and Output

This chapter tells you about input to and output from ILOG CPLEX. It covers the following topics:

- *Understanding File Formats* on page 156;
- *Using Concert XML Extensions* on page 159
- *Using Concert csvReader* on page 160;
- *Managing Log Files* on page 161;
- *Controlling Message Channels* on page 162.

> ***Note:*** *There are platforms limit the size of files that they can read. If you have created a problem file on one platform, and you find that you are unable to read the problem on another platform, consider whether the platform where you are trying to read the file suffers from such a limit on file size. ILOG CPLEX may be unable to open your problem file due to the size of the file being greater than the platform limit.*

Understanding File Formats

The reference manual *ILOG CPLEX File Formats* documents the file formats that ILOG CPLEX supports. The following sections cover programming considerations about widely used file formats.

- *Working with LP Files* on page 156;
- *Working with MPS Files* on page 157;
- *Converting File Formats* on page 158.

Working with LP Files

LP files are row-oriented so you can look at a problem as you enter it in a naturally and intuitively algebraic way. However, ILOG CPLEX represents a problem internally in a column-ordered format. This difference between the way ILOG CPLEX accepts a problem in LP format and the way it stores the problem internally may have an impact on *memory use* and on the *order* in which *variables* are displayed on screen or in files.

Variable Order and LP Files

As ILOG CPLEX reads an LP format file by rows, it adds columns as it encounters them in a row. This convention will have an impact on the order in which variables are named and displayed. For example, consider this problem:

Maximize $2x_2 + 3x_3$

subject to

$$-x_1 + x_2 + x_3 \leq 20$$
$$x_1 - 3x_2 + x_3 \leq 30$$

with these bounds

$$0 \leq x_1 \leq 40$$
$$0 \leq x_2 \leq +\infty$$
$$0 \leq x_3 \leq +\infty$$

Since ILOG CPLEX reads the objective function as the first row, the two columns appearing there will become the first two variables. When the problem is displayed or rewritten into

another LP file, the variables there will appear in a different order within each row. In this example, if you execute the command `display problem all`, you will see this:

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

That is, `x1` appears at the end of each constraint in which it has a nonzero coefficient. Also, while re-ordering like this does not affect the optimal objective function value of the problem, if there exist alternate optimal solutions at this value, then the different order of the variables could result in a change in the solution path of the algorithm, and there may be noticeable variation in the solution values of the individual variables.

Working with MPS Files

The ILOG CPLEX MPS file reader is highly compatible with existing modeling systems. There is generally no need to modify existing problem files to use them with ILOG CPLEX. However, there are ILOG CPLEX-specific conventions that may be useful for you to know. This section explains those conventions, and the reference manual *ILOG CPLEX File Formats* documents the MPS format more fully.

Free Rows in MPS Files

In an MPS file, ILOG CPLEX selects the first free row or N-type row as the objective function, and it discards all subsequent free rows unless it is instructed otherwise by an `OBJNAME` section in the file. To retain free rows in an MPS file, reformulate them as equality rows with an additional free variable. For example, replace the free row `x + y` by the equality row `x + y - s = 0` where `s` is free. Generally, the ILOG CPLEX presolver will remove rows like that before optimization so they will have no impact on performance.

Ranged Rows in MPS Files

To handle ranged rows, ILOG CPLEX introduces a temporary range variable, creates appropriate bounds for this variable, and changes the sense of the row to an equality (that is, MPS type EQ). The added range variables will have the same name as the ranged row with the characters `Rg` prefixed. When ILOG CPLEX generates solution reports, it removes these temporary range variables from the constraint matrix.

Extra Rim Vectors in MPS Files

The MPS format allows multiple right-hand sides (RHSs), multiple bounds, and multiple range vectors. It also allows extra free rows. Together, these features are known as *extra rim vectors*. By default, the ILOG CPLEX MPS reader selects the first RHS, bound, and range

definitions that it finds. The first free row (that is, N-type row) becomes the objective function, and the remaining free rows are discarded. The extra rim data are also discarded.

Naming Conventions in MPS Files

ILOG CPLEX accepts any noncontrol-character within a name. However, ILOG CPLEX recognizes blanks (that is, spaces) as delimiters, so you must avoid them in names. You should also avoid $ (dollar sign) and * (asterisk) as characters in names because they normally indicate a comment within a data record.

Error Checking in MPS Files

Fairly common problems in MPS files include split vectors, unnamed columns, and duplicated names. ILOG CPLEX checks for these conditions and reports them. If repeated rows or columns occur in an MPS file, ILOG CPLEX reports an error and stops reading the file. You can then edit the MPS file to correct the source of the problem.

Saving Modified MPS Files

You may often want to save a modified MPS file for later use. To that end, ILOG CPLEX will write out a problem exactly as it appears in memory. All your revisions of that problem will appear in the new file. One potential area for confusion occurs when a maximization problem is saved. Since MPS conventionally represents all problems as minimizations, ILOG CPLEX reverses the sign of the objective-function coefficients when it writes a maximization problem to an MPS file. When you read and optimize this new problem, the values of the variables will be valid for the original model. However, since the problem has been converted from a maximization to the equivalent minimization, the objective, dual, and reduced-cost values will have reversed signs.

Converting File Formats

MPS, Mathematical Programming System, an industry-standard format based on ASCII-text has historically been restricted to a fixed format in which data fields were limited to eight characters and specific fields had to appear in specific columns on specific lines. ILOG CPLEX supports extensions to MPS that allow more descriptive names (that is, more than eight characters), greater accuracy for numerical data, and greater flexibility in data positions.

Most MPS files in fixed format conform to the ILOG CPLEX extensions and thus can be read by the ILOG CPLEX MPS reader without error. However, the ILOG CPLEX MPS reader will *not* accept the following conventions:

- blank space within a name;
- blank lines;
- missing fields (such as bound names and right-hand side names);
- extraneous, uncommented characters;

- blanks in lieu of repeated name fields, such as bound vector names and right-hand side names.

You can convert fixed-format MPS files that contain those conventions into acceptable ILOG CPLEX-extended MPS files. To do so, use the `convert` utility supplied in the standard distribution of ILOG CPLEX. The `convert` utility removes unreadable features from fixed-format MPS, BAS, SOS, and ORD files. It runs from the operating system prompt of your platform. Here is the syntax of the convert utility:

```
convert -option inputfilename outputfilename
```

Your command must include an input-file name and an output-file name; they must be different from each other. The options, summarized in Table 6.1, indicate the file type. You may specify only one option. If you do not specify an option, ILOG CPLEX attempts to deduce the file type from the extension in the file name.

Table 6.1 *Options for the* `convert` *Utility and Corresponding File Extensions*

Option	File type	File extension
`-m`	MPS (Mathematical Programming System)	`.mps`
`-s`	SOS (Special Ordered Set)	`.sos`
`-b`	BAS (basis file according to MPS conventions)	`.bas`
`-o`	ORD (priority orders)	`.ord`

Using Concert XML Extensions

Concert Technology for C++ users offers a suite of classes for serializing ILOG CPLEX models (that is, instances of `IloModel`) and solutions (that is, instances of `IloSolution`) through XML. The *Concert Technology C++ API Reference Manual* documents the XML serialization API in the group `optim.concert.xml`. That group includes these classes:

- `IloXmlContext` allows you to serialize an instance of `IloModel` or `IloSolution`. This class offers methods for reading and writing a model, a solution, or both a model and a solution together. There are examples of how to use this class in the reference manual.
- `IloXmlInfo` offers methods that enable you to validate the XML serialization of elements, such as numeric arrays, integer arrays, variables, and other extractables from your model or solution.

- `IloXmlReader` creates a reader in an environment (that is, in an instance of `IloEnv`). This class offers methods to check runtime type information (RTTI), to recognize hierarchic relations between objects, and to access attributes of objects in your model or solution.
- `IloXmlWriter` creates a writer in an environment (that is, in an instance of `IloEnv`). This class offers methods to access elements and to convert their types as needed in order to serialize elements of your model or solution.

> ***Note:*** *There is a fundamental difference between writing an XML file of a model and writing an LP/MPS/SAV file of the same extracted model. If the model contains piecewise linear elements (PWL), or other nonlinear features, the XML file will represent the model as such. In contrast, the LP/MPS/SAV file will represent only the tranformed model. That transformed model obscures these nonlinear features because of the automatic transformation that took place.*

Using Concert csvReader

CSV is a file format consisting of lines of comma-separated values in ordinary ASCII text. Concert Technology for C++ users provides classes adapted to reading data into your application from a CSV file. The constructors and methods of these classes are documented more fully in the *Concert Technology C++ Reference Manual*.

- `IloCsvReader`

  An object of this class is capable of reading data from a CSV file and passing the data to your application. There are methods in this class for recognizing the first line of the file as a header, for indicating whether or not to cache the data, for counting columns, for counting lines, for accessing lines by number or by name, for designating special characters, for indicating separators, and so forth.

- `IloCsvLine`

  An object of this class represents a line of a CSV file. The constructors and methods of this class enable you to designate special characters, such as a decimal point, separator, line ending, and so forth.

- `IloCsvReader::Iterator`

  An object of this embedded class is an iterator capable of accessing data in a CSV file line by line. This iterator is useful, for example, in programming loops of your application, such as `while`-statements.

Managing Log Files

As ILOG CPLEX is working, it can record messages to a log file. By default, the Interactive Optimizer creates the log file in the directory where it is running, and it names the file `cplex.log`. If such a file already exists, ILOG CPLEX adds a line indicating the current time and date and then appends new information to the end of the existing file. That is, it does not overwrite the file, and it distinguishes different sessions within the log file. By default, there is no log file for Component Library applications.

You can locate the log file where you like, and you can rename it. Some users, for example, like to create a specifically named log file for each session. Also you can close the log file in case you do not want ILOG CPLEX to record messages to its default log file.

The following sections show you the commands for creating, renaming, relocating, and closing a log file.

Creating, Renaming, Relocating Log Files

- In the Interactive Optimizer, use the command `set logfile filename`, substituting the name you prefer for the log file. In other words, use this command to rename or relocate the default log file.
- From the Callable Library, first use the routine `CPXfopen` to open the target file; then use the routine `CPXsetlogfile`. The *ILOG CPLEX Reference Manual* documents both routines.
- From Concert, use the `setOut` method to send logging output to the specified output stream.

Closing Log Files

- If you do not want ILOG CPLEX to record messages in a log file, then you can close the log file from the Interactive Optimizer with the command `set logfile *`.
- By default, routines from the Callable Library do not write to a log file. However, if you want to close a log file that you created by a call to `CPXsetlogfile`, call `CPXsetlogfile` again, and this time, pass a `NULL` pointer as its second argument.
- From Concert, use the `setOut` method with `env.getNullStream` as argument, where `env` is an `IloEnv` object, to stop sending logging output to an output stream.

Controlling Message Channels

In both the Interactive Optimizer and the Callable Library, there are message channels that enable you to direct output from your application as you prefer. In the Interactive Optimizer, these channels are defined by the command `set output channel` with its options as listed in Table 6.2. In the Callable Library, there are routines for managing message channels, in addition to parameters that you can set. The following sections offer more details about these ideas:

- *Parameter for Output Channels* on page 162;
- *Callable Library Routines for Message Channels* on page 163;
- *Example: Callable Library Message Channels* on page 164.

Parameter for Output Channels

Besides the log-file parameter, Interactive Optimizer and the Callable Library offer you output-channel parameters to give you finer control over when and where messages appear in the Interactive Optimizer. Output-channel parameters indicate whether output should or should not appear on screen. They also allow you to designate log files for message channels. The output-channel parameters do not affect the log-file parameter, so it is customary to use the command `set logfile` before the command `set output channel value1 value2`.

In the output-channel command, you can specify a `channel` to be one of `dialog`, `errors`, `logonly`, `results`, or `warnings`. Table 6.2 summarizes the information carried over each channel.

Table 6.2 *Options for the Output-Channel Command*

Channel	Information
`dialog`	messages related to interactive use; e.g., prompts, help messages, greetings
`errors`	messages to inform user that operation could not be performed and why
`logonly`	message to record only in file (not on screen) e.g., multiline messages
`results`	information explicitly requested by user; state, change, progress information
`warnings`	messages to inform user request was performed but unexpected condition may result

The option `value2` lets you specify a file name to redirect output from a channel.

Also in that command, `value1` allows you to turn on or off output to the screen. When `value1` is `y`, output is directed to the screen; when its value is `n`, output is not directed to the

screen. Table 6.3 summarizes which channels direct output to the screen by default. If a channel directs output to the screen by default, you can leave `value1` blank to get the same effect as `set output channel y`.

Table 6.3 *Channels Directing Output to Screen or to a File*

Channel	Default value1	Meaning
`dialog`	`y`	blank directs output to screen but not to a file
`errors`	`y`	blank directs output to screen and to a file
`logonly`	`n`	blank directs output only to a file, not to screen
`results`	`y`	blank directs output to screen and to a file
`warnings`	`y`	blank directs output to screen and to a file

Callable Library Routines for Message Channels

Interactive Optimizer and the Callable Library define several message channels for flexible control over message output:

- `cpxresults` for messages containing status and progress information;
- `cpxerror` for messages issued when a task cannot be completed;
- `cpxwarning` for messages issued when a nonfatal difficulty is encountered; or when an action taken may have side-effects; or when an assumption made may have side-effects;
- `cpxlog` for messages containing information that would not conventionally be displayed on screen but could be useful in a log file.

Output messages flow through message channels to destinations. Message channels are associated with destinations through their destination list. Messages from routines of the ILOG CPLEX Callable Library are assigned internally to one of those predefined channels. Those default channels are C pointers to ILOG CPLEX objects; they are initialized by `CPXopenCPLEX`; they are *not* global variables. Your application accesses these objects by calling the routine `CPXgetchannels`. You can use these predefined message channels for your own application messages. You can also define new channels.

An application using routines from the ILOG CPLEX Callable Library produces no output messages unless the application specifies message handling instructions through one or more calls to the message handling routines of the Callable Library. In other words, the destination list of each channel is initially empty.

Messages from multiple channels may be sent to one destination. All predefined ILOG CPLEX channels can be directed to a single file by a call to `CPXsetlogfile`.

Similarly, all predefined ILOG CPLEX channels except `cpxlog` can be directed to the screen by the `CPX_PARAM_SCRIND` parameter. For a finer level of control, or to define destinations for application-specific messages, use the following message handling routines, all documented in the *ILOG CPLEX Reference Manual*:

- `CPXmsg` writes a message to a predefined channel;
- `CPXflushchannel` flushes a channel to its associated destination;
- `CPXdisconnectchannel` flushes a channel and clears its destination list;
- `CPXdelchannel` flushes a channel, clears its destination list, frees memory for that channel;
- `CPXaddchannel` adds a channel;
- `CPXaddfpdest` adds a destination file to the list of destinations associated with a channel;
- `CPXdelfpdest` deletes a destination from the destination list of a channel;
- `CPXaddfuncdest` adds a destination function to a channel;
- `CPXdelfuncdest` deletes a destination function to a channel;

Once channel destinations are established, messages can be sent to multiple destinations by a single call to a message-handling routine.

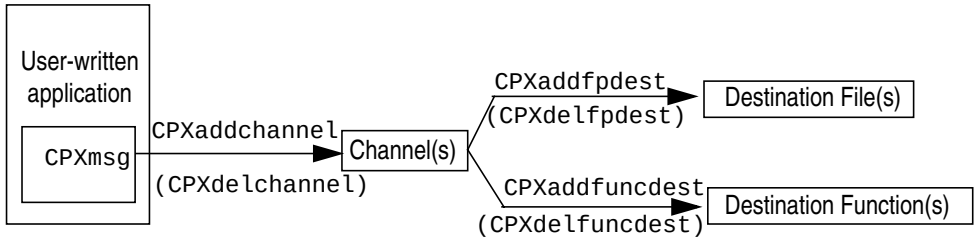

Figure 6.1 *ILOG CPLEX Message Handling Routines*

Example: Callable Library Message Channels

This example shows you how to use the ILOG CPLEX message handler from the Callable Library. It captures all messages generated by ILOG CPLEX and displays them on screen along with a label indicating which channel sent the message. It also creates a user channel to receive output generated by the program itself. The user channel accepts user-generated messages, displays them on screen with a label, and records them in a file without the label.

This example derives from `lpex1.c`, a program in the ILOG CPLEX *Getting Started* manual. There are a few differences between the two examples:

- In this example, the function `ourmsgfunc` (rather than the C functions `printf` or `fprintf(stderr, . . .)`) manages all output. The program itself or `CPXmsg` from the ILOG CPLEX Callable Library calls `ourmsgfunc`. In fact, `CPXmsg` is a replacement for `printf`, allowing a message to appear in *more than one place*, for example, both on screen and in a file.

  Only *after* you initialize the ILOG CPLEX environment by calling `CPXopenCPLEX` can you call `CPXmsg`. And only *after* you call `CPXgetchannels` can you use the default ILOG CPLEX channels. Therefore, calls to `ourmsgfunc` print directly any messages that occur *before* the program gets the address of `cpxerror` (a channel). After a call to `CPXgetchannels` gets the address of `cpxerror`, and after a call to `CPXaddfuncdest` associates the message function `ourmsgfunc` with `cpxerror`, then error messages are generated by calls to `CPXmsg`.

  After the `TERMINATE:` label, any error must be generated with care in case the error message function has not been set up properly. Thus, `ourmsgfunc` is also called directly to generate any error messages there.

- A call to the ILOG CPLEX Callable Library routine `CPXaddchannel` initializes the channel `ourchannel`. The Callable Library routine `fopen` opens the file `lpex5.out` to accept solution information. A call the ILOG CPLEX Callable Library routine `CPXaddfpdest` associates that file with that channel. Solution information is also displayed on screen since `ourmsgfunc` is associated with that new channel, too. Thus in the loops near the end of `main`, when the solution is printed, only one call to `CPXmsg` suffices to put the output both on screen and into the file. A call to `CPXdelchannel` deletes `ourchannel`.

- Although `CPXcloseCPLEX` will automatically delete file- and function-destinations for channels, it is a good practice to call `CPXdelfpdest` and `CPXdelfuncdest` at the end of your programs.

Complete Program: lpex5.c

The complete program, `lpex5.c`, appears here and online in the standard distribution.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/lpex5.c                                              */
/*  Version 9.0                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2003 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* lpex5.c - Illustrating the CPLEX message handler
   This is a modification of lpex1.c .
   We'll label each channels output with its name, and
```

```
   also put our own output into a file.  Note that the labels
   may cause the output to go past 80 characters per line. */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

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

#include <string.h>

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

static int
   populatebycolumn  (CPXENVptr env, CPXLPptr lp);

static void CPXPUBLIC
   ourmsgfunc     (void *handle, const char *message);



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

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


int
main (void)
{
   char     probname[16];  /* Problem name is max 16 characters */

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

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


   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status;
   int           i, j;
   int           cur_numrows, cur_numcols;
   char          errmsg[1024];

   CPXCHANNELptr  cpxerror   = NULL;
   CPXCHANNELptr  cpxwarning = NULL;
   CPXCHANNELptr  cpxresults = NULL;
```

```
CPXCHANNELptr  ourchannel = NULL;

char *errorlabel = "cpxerror";
char *warnlabel  = "cpxwarning";
char *reslabel   = "cpxresults";
char *ourlabel   = "Our Channel";

CPXFILEptr fpout  = NULL;


/* Initialize the CPLEX environment */

env = CPXopenCPLEX (&status);

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

/* Since the message handler is yet to be set up, we'll call our
   messaging function directly to print out any errors  */

if ( env == NULL ) {
   /* The message argument for ourmsgfunc must not be a constant,
      so copy the mesage to a buffer. */
   strcpy (errmsg, "Could not open CPLEX environment.\n");
   ourmsgfunc ("Our Message", errmsg);
   goto TERMINATE;
}

/* Now get the standard channels.  If an error, just call our
   message function directly. */

status = CPXgetchannels (env, &cpxresults, &cpxwarning, &cpxerror, NULL);
if ( status ) {
   strcpy (errmsg, "Could not get standard channels.\n");
   ourmsgfunc ("Our Message", errmsg);
   CPXgeterrorstring (env, status, errmsg);
   ourmsgfunc ("Our Message", errmsg);
   goto TERMINATE;
}

/* Now set up the error channel first.  The label will be "cpxerror" */

status = CPXaddfuncdest (env, cpxerror, errorlabel, ourmsgfunc);
if ( status ) {
   strcpy (errmsg, "Could not set up error message handler.\n");
   ourmsgfunc ("Our Message", errmsg);
   CPXgeterrorstring (env, status, errmsg);
   ourmsgfunc ("Our Message", errmsg);
}

/* Now that we have the error message handler set up, all CPLEX
   generated errors will go through ourmsgfunc.  So we don't have
   to use CPXgeterrorstring to determine the text of the message.
   We can also use CPXmsg to do any other printing.  */
```

```
status = CPXaddfuncdest (env, cpxwarning, warnlabel, ourmsgfunc);
if ( status ) {
   CPXmsg (cpxerror, "Failed to set up handler for cpxwarning.\n");
   goto TERMINATE;
}

status = CPXaddfuncdest (env, cpxresults, reslabel, ourmsgfunc);
if ( status ) {
   CPXmsg (cpxerror, "Failed to set up handler for cpxresults.\n");
   goto TERMINATE;
}

/* Now turn on the iteration display. */

status = CPXsetintparam (env, CPX_PARAM_SIMDISPLAY, 2);
if ( status ) {
   CPXmsg (cpxerror, "Failed to turn on simplex display level.\n");
   goto TERMINATE;
}

/* Create the problem. */

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

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

if ( lp == NULL ) {
   CPXmsg (cpxerror, "Failed to create LP.\n");
   goto TERMINATE;
}

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

status = populatebycolumn (env, lp);

if ( status ) {
   CPXmsg (cpxerror, "Failed to populate problem data.\n");
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */

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

status = CPXsolution (env, lp, &solstat, &objval, x, pi, slack, dj);
if ( status ) {
   CPXmsg (cpxerror, "Failed to obtain solution.\n");
```

```
   goto TERMINATE;
}


/* Write the output to the screen.  We will also write it to a
   file as well by setting up a file destination and a function
   destination. */

ourchannel = CPXaddchannel (env);
if ( ourchannel == NULL ) {
   CPXmsg (cpxerror, "Failed to set up our private channel.\n");
   goto TERMINATE;
}

fpout = CPXfopen ("lpex5.msg", "w");
if ( fpout == NULL ) {
   CPXmsg (cpxerror, "Failed to open lpex5.msg file for output.\n");
   goto TERMINATE;
}
status = CPXaddfpdest (env, ourchannel, fpout);
if ( status ) {
   CPXmsg (cpxerror, "Failed to set up output file destination.\n");
   goto TERMINATE;
}

status = CPXaddfuncdest (env, ourchannel, ourlabel, ourmsgfunc);
if ( status ) {
   CPXmsg (cpxerror, "Failed to set up our output function.\n");
   goto TERMINATE;
}

/* Now any message to channel ourchannel will go into the file
   and into the file opened above. */

CPXmsg (ourchannel, "\nSolution status = %d\n", solstat);
CPXmsg (ourchannel, "Solution value  = %f\n\n", objval);

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

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

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

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

status = CPXwriteprob (env, lp, "lpex5.lp", NULL);
if ( status ) {
```

```
      CPXmsg (cpxerror, "Failed to write LP to disk.\n");
      goto TERMINATE;
   }

TERMINATE:

   /* First check if ourchannel is open */

   if ( ourchannel != NULL ) {
      int  chanstat;
      chanstat = CPXdelfuncdest (env, ourchannel, ourlabel, ourmsgfunc);
      if ( chanstat ) {
         strcpy (errmsg, "CPXdelfuncdest failed.\n");
         ourmsgfunc ("Our Message", errmsg);
         if (!status)  status = chanstat;
      }
      if ( fpout != NULL ) {
         chanstat = CPXdelfpdest (env, ourchannel, fpout);
         if ( chanstat ) {
            strcpy (errmsg, "CPXdelfpdest failed.\n");
            ourmsgfunc ("Our Message", errmsg);
            if (!status)  status = chanstat;
         }
         CPXfclose (fpout);
      }

      CPXdelchannel (env, &ourchannel);
   }

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

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         strcpy (errmsg, "CPXfreeprob failed.\n");
         ourmsgfunc ("Our Message", errmsg);
      }
   }

   /* Now delete our function destinations from the 3 CPLEX channels. */
   if ( cpxresults != NULL ) {
      int  chanstat;
      chanstat = CPXdelfuncdest (env, cpxresults, reslabel, ourmsgfunc);
      if ( chanstat && !status ) {
         status = chanstat;
         strcpy (errmsg, "Failed to delete cpxresults function.\n");
         ourmsgfunc ("Our Message", errmsg);
      }
   }

   if ( cpxwarning != NULL ) {
      int  chanstat;
      chanstat = CPXdelfuncdest (env, cpxwarning, warnlabel, ourmsgfunc);
      if ( chanstat && !status ) {
         status = chanstat;
         strcpy (errmsg, "Failed to delete cpxwarning function.\n");
         ourmsgfunc ("Our Message", errmsg);
      }
```

```
   }

   if ( cpxerror != NULL ) {
      int  chanstat;
      chanstat = CPXdelfuncdest (env, cpxerror, errorlabel, ourmsgfunc);
      if ( chanstat && !status ) {
         status = chanstat;
         strcpy (errmsg, "Failed to delete cpxerror function.\n");
         ourmsgfunc ("Our Message", errmsg);
      }
   }

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

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

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

      if ( status ) {
         strcpy (errmsg, "Could not close CPLEX environment.\n");
         ourmsgfunc ("Our Message", errmsg);
         CPXgeterrorstring (env, status, errmsg);
         ourmsgfunc ("Our Message", errmsg);
      }
   }

   return (status);

}  /* END main */


/* This function builds by column the linear program:

      Maximize
       obj: x1 + 2 x2 + 3 x3
      Subject To
       c1: - x1 + x2 + x3 <= 20
       c2: x1 - 3 x2 + x3 <= 30
      Bounds
       0 <= x1 <= 40
      End
 */

static int
populatebycolumn (CPXENVptr env, CPXLPptr lp)
{
   int      status    = 0;
   double   obj[NUMCOLS];
   double   lb[NUMCOLS];
   double   ub[NUMCOLS];
   char     *colname[NUMCOLS];
   int      matbeg[NUMCOLS];
   int      matind[NUMNZ];
   double   matval[NUMNZ];
```

```
   double   rhs[NUMROWS];
   char     sense[NUMROWS];
   char     *rowname[NUMROWS];

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

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

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

   rowname[0] = "c1";
   sense[0]   = 'L';
   rhs[0]     = 20.0;

   rowname[1] = "c2";
   sense[1]   = 'L';
   rhs[1]     = 30.0;

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

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

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

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

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

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

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

   colname[0] = "x1"; colname[1] = "x2";      colname[2] = "x3";

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

TERMINATE:

   return (status);

}  /* END populatebycolumn */


/* For our message functions, we will interpret the handle as a pointer
 * to a string, which will be the label for the channel.  We'll put
 * angle brackets <> around the message so its clear what the function is
 * sending to us.  We'll place the newlines that appear at the end of
 * a message after the > bracket.  The 'message' argument must not be
 * a constant, since it is changed by this function.
 */
```

```
static void CPXPUBLIC
ourmsgfunc (void *handle, const char *msg)
{
   char  *label;
   int   lenstr;
   int   flag = 0;
   char  *message = (char *)msg;

   lenstr = strlen(message);
   if ( message[lenstr-1] == '\n' ) {
      message[lenstr-1] = '\0';
      flag = 1;
   }

   label = (char *) handle;
   printf ("%-15s: <%s>", label, message);
   if (flag) putchar('\n');

   /* If we clobbered the '\n', we need to put it back */

   if ( flag )  message[lenstr-1] = '\n';

} /* END ourmsgfunc */
```

C H A P T E R

7

Licensing an Application

This chapter tells you about CPLEX runtime development and licensing procedures.

ILOG CPLEX uses the standard ILOG License Manager (ILM). The *ILOG License Manager* online documentation comes with your software distribution. It documents ILM access keys (or *keys*, for short) in more detail. This chapter shows you how to write applications that use ILM runtime access keys.

An ILOG CPLEX runtime license costs significantly less money than a regular development license. However, its use is restricted to applications created by a particular developer or company. In order to distinguish runtime access keys from development keys (as well as runtime keys for applications created by other companies), you need to call an additional routine in your source code before initializing the CPLEX environment.

This chapter includes the following topics:

- *Types of ILM Runtime Licenses* on page 176
- *Routines and Methods Used for Licensing* on page 176
- *Examples* on page 177
- *Summary* on page 179

Types of ILM Runtime Licenses

ILM runtime licenses come in two forms: file-based and memory-based.

File-Based RTNODE, RTSTOKEN or TOKEN Keys.

These are a file-based access key that is tied to a particular computer or server. Refer to the *ILOG License Manager online documentation*, which can be found in your distribution of the product, for information about how to establish the file containing the key. You must communicate the location of this file to your application. In order to avoid potential conflicts with other runtime applications, it is a good idea to put the key in a directory specific to your application by using one of the following:

- the C routine `CPXputenv` from the Callable Library
- the C routine `CPXputenv` from the Callable Library in the C++ API of Concert Technology
- the method `IloCplex.putEnv` in the Java API of Concert Technology
- the method `Cplex.PutEnv` in the C#.NET API of Concert Technology

These are the most commonly used runtime licenses.

Memory-Based RUNTIME Keys.

These involve passing some information in the memory of your program to ILM. No files containing access keys are involved. Rather, you set the key in your program and pass it to ILM by calling one of the following:

- the C routine `CPXRegisterLicense` from the Callable Library
- the C routine `CPXRegisterLicense` from the Callable Library in the C++ API of Concert Technology
- the method `IloCplex.registerLicense` in the Java API of Concert Technology
- the method `Cplex.RegisterLicense` in the C#.NET API of Concert Technology

Routines and Methods Used for Licensing

All ILOG CPLEX applications either call the routine `CPXopenCPLEX` to establish the CPLEX environment, or use the appropriate constructor (`IloCplex` in the C++ and Java API or `Cplex` in the C#.NET API) to initialize ILOG CPLEX for use with Concert Technology. Until either `CPXopenCPLEX` is called or the `IloCplex` object exists, few

ILOG CPLEX routines or methods operate. In addition to allocating the environment, `CPXopenCPLEX` performs license checks, as do the constructors for Concert Technology. For development licenses, no additional licensing steps are required. For runtime licenses, your application first needs to provide some additional licensing information before the call to `CPXopenCPLEX` or the use of a constructor.

- For `RTNODE`, `RTSTOKEN` and `TOKEN` keys, this requires calling the `CPXputenv` routine from the Callable Library and C++ API of Concert Technology, or the `IloCplex.putenv` static method from the Java API, or `Cplex.PutEnv` from the C#.NET API, to specify the location of the key through the `ILOG_LICENSE_FILE` environment variable.
- For memory-based `RUNTIME` keys, this requires calling the `CPXRegisterLicense` routine for Callable Library and C++ users, or the static method `IloCplex.registerLicense` for Java users, or the static method `Cplex.RegisterLicense` for C#.NET users, to pass the `RUNTIME` key to ILM.

Documentation of the routines `CPXputenv` and `CPXRegisterLicense` is in the *ILOG CPLEX Callable Library Reference Manual*; documentation of `IloCplex.putenv` and `IloCplex.registerLicense` is in the *ILOG CPLEX Java API Reference Manual*; documentation of `Cplex.PutEnv` and `Cplex.RegisterLicense` is in the *ILOG CPLEX C#.NET API Reference Manual.*

Examples

Here are some code samples that illustrate the use of those runtime license routines and methods. The first example illustrates the routine `CPXputenv` when opening the CPLEX environment.

> ***Notes:*** *This example assumes a Microsoft Windows file directory structure that requires an additional backslash when specifying the path of the file containing the key. It also assumes that the application uses an environment variable called* `MYAPP_HOME` *to identify the directory in which it was installed.*
>
> *The string argument to* `CPXputenv` *must remain active throughout the time ILOG CPLEX is active; the best way to do this is to* `malloc` *the string.*

CPXputenv Routine

```
char *inststr = NULL;
char *envstr  = NULL;

/* Initialize the CPLEX environment */

envstr = (char *) malloc (256);
```

```
if ( envstr == NULL ) {
   fprintf (stderr, "Memory allocation for CPXputenv failed.\n");
   status = FAIL;
   goto TERMINATE;
}
else {
   inststr = (char *)  getenv("MYAPP_HOME");
   if ( inststr == NULL ) {
      fprintf (stderr, "Unable to find installation directory.\n");
      status = FAIL;
      goto TERMINATE;
   }
   strcpy (envstr, "ILOG_LICENSE_FILE=");
   strcat (envstr, inststr);
   strcat (envstr, "\\license\\access.ilm");
   CPXputenv (envstr);
}

env = CPXopenCPLEX (&status);
```

The putenv Method for Java Users

Here is an example using Concert Technology for Java users:

```
IloCplex.putenv("ILOG_LICENSE_FILE=\\license\\access.ilm");
try {
   cplex = new IloCplex();
}
catch (IloException e) {
   System.err.println("Exception caught for runtime license:" + e);
}
```

CPXRegisterLicense Routine

The following is an example showing how to use the routine `CPXRegisterLicense`.

```
static char *ilm_license=\
 "LICENSE ILOG Incline\n\
  RUNTIME CPLEX      9.000 21-Apr-2004 R81GM34ECZTS N IRIX , options: m ";
static int ilm_license_signature=2756133;

   CPXENVptr     env = NULL;
   int           status;

   /* Initialize the CPLEX environment */

    status = CPXRegisterLicense (ilm_license, ilm_license_signature);
    if ( status != 0) {
       fprintf (stderr, "Could not register CPLEX license, status %d.\n",
                status);
       goto TERMINATE;
```

```
    }
    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;
    }
```


The registerLicense Method for Java Users

Here is an example for Java users applying `IloCplex.registerLicense`:

```
static String ilm_CPLEX_license=
"LICENSE ILOG Test\n RUNTIME CPLEX      9.000 021-Apr-2004 R81GM34ECZTS N IRIX
,
options: m ";
static int ilm_CPLEX_license_signature=2756133;

public static void main(String[] args) {

   try {
      IloCplex.registerLicense(ilm_CPLEX_license, ilm_CPLEX_license_signature);
      IloCplex cplex = new IloCplex();
   }
   catch (IloException e) {
      System.err.println("Exception caught for runtime license:" + e);
   }
}
```

Summary

ILM runtime license keys come in two forms. Users of file-based `RTNODE`, `RTSTOKEN` and `TOKEN` keys should use the routine `CPXputenv` or the method `IloCplex.putenv` or the method `Cplex.PutEnv` to identify the location of the license file. Users of memory-based `RUNTIME` keys should use the routine `CPXRegisterLicense` or the method `IloCplex.registerLicense` or the method `Cplex.RegisterLicense` to pass the key to the ILOG License Manager embedded inside the CPLEX Component Libraries. Refer to the *ILOG License Manager online documentation* that comes with your software for additional information about activating and maintaining ILM license keys.

Part III

Continuous Optimization

This part focuses on algorithmic considerations about the ILOG CPLEX optimizers that solve problems formulated in terms of **continuous** variables. While ILOG CPLEX is delivered with default settings that enable you to solve many problems without changing parameters, this part also documents features that you can customize for your application. This part contains:

- *Solving LPs: Simplex Optimizers* on page 183
- *Solving LPs: Barrier Optimizer* on page 225
- *Solving Network-Flow Problems* on page 245
- *Solving Problems with a Quadratic Objective (QP)* on page 265
- *Solving Problems with Quadratic Constraints (QCP)* on page 295

C H A P T E R

8

Solving LPs: Simplex Optimizers

The preceding chapters have focused on the details of writing applications that model optimization problems and access the solutions to those problems, with minimal attention to the optimizer that solves them, because most models are solved well by the default optimizers provided by ILOG CPLEX. For instances where a user wishes to exert more direct influence over the solution process, ILOG CPLEX provides a number of features that may be of interest.

This chapter and the next one tell you more about solving linear programs with ILOG CPLEX using the LP optimizers. This chapter emphasizes primal and dual simplex optimizers. It contains sections about:

- *Choosing an Optimizer for Your LP Problem* on page 184
- *Tuning LP Performance* on page 187
- *Diagnosing Performance Problems* on page 195
- *Diagnosing LP Infeasibility* on page 200
- *Example: Using a Starting Basis in an LP Problem* on page 215

Choosing an Optimizer for Your LP Problem

ILOG CPLEX offers several different optimizers for linear programming problems. Each of these optimizers is available whether you call ILOG CPLEX from within your own application using Concert or the Callable Library, or you use the Interactive Optimizer.

The choice of LP optimizer in ILOG CPLEX can be specified through a parameter, named `CPX_PARAM_LPMETHOD` in the Callable Library, and named `LPMETHOD` in the Interactive Optimizer. In both the C++ and Java versions of Concert, the LP method is controlled by the `RootAlg` parameter (which also controls related aspects of QP and MIP solutions, as explained in the corresponding chapters of this manual). In this chapter, this parameter will be referred to uniformly as `LPMethod`.

The `LPMethod` parameter determines which optimizer will be used when you solve a model in one of the following ways:

- `cplex.solve` (Concert)
- `CPXlpopt` (Callable Library)
- `optimize` (Interactive Optimizer)

The choices for `LPMethod` are summarized in Table 8.1.

Table 8.1 *Settings of the LPMethod Parameter for Choosing an Optimizer*

Setting of LPMethod	Meaning	See Section
0	Default Setting	*Automatic Selection of Optimizer* on page 185
1	Primal Simplex	*Primal Simplex Optimizer* on page 186
2	Dual Simplex	*Dual Simplex Optimizer* on page 185
3	Network Simplex	*Network Optimizer* on page 186
4	Barrier	*Barrier Optimizer* on page 186
5	Sifting	*Sifting Optimizer* on page 186
6	Concurrent Dual, Barrier, and Primal	*Concurrent Optimizer* on page 187

The symbolic names for these settings in an application program are summarized in Table 8.2.

Table 8.2 *Symbolic Names for LP Solution Methods*

	Concert C++	Concert Java	Concert.NET	Callable Library
0	IloCplex::AutoAlg	IloCplex.Algorithm.Auto	Cplex.Auto	CPX_ALG_AUTOMATIC
1	IloCplex::Primal	IloCplex.Algorithm.Primal	Cplex.Primal	CPX_ALG_PRIMAL
2	IloCplex::Dual	IloCplex.Algorithm.Dual	Cplex.Dual	CPX_ALG_DUAL
3	IloCplex::Network	IloCplex.Algorithm.Network	Cplex.Network	CPX_ALG_NET
4	IloCplex::Barrier	IloCplex.Algorithm.Barrier	Cplex.Barrier	CPX_ALG_BARRIER
5	IloCplex::Sifting	IloCplex.Algorithm.Sifting	Cplex.Sifting	CPX_ALG_SIFTING
6	IloCplex::Concurrent	IloCplex.Algorithm.Concurrent	Cplex.Concurrent	CPX_ALG_CONCURRENT

Automatic Selection of Optimizer

The default `Automatic` setting of `LPMethod` lets ILOG CPLEX determine the algorithm to use to optimize your problem. Most models are solved well with this setting, and this is the recommended option except when you have a compelling reason to tune performance for a particular class of model.

On a serial computer, or on a parallel computer where only one thread will be invoked, the automatic setting will in most cases choose the dual simplex optimizer. An exception to this rule is when an advanced basis is present that is ascertained to be primal feasible; in that case, primal simplex will be called.

On a computer where parallel threads are available to ILOG CPLEX, the automatic setting results in the concurrent optimizer being called. An exception is when an advanced basis is present; in that case, it will behave as the serial algorithm would.

Dual Simplex Optimizer

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

Primal Simplex Optimizer

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

Network Optimizer

If a major part of your problem is structured as a network, then the ILOG CPLEX Network Optimizer may have a positive impact on performance. The ILOG CPLEX Network Optimizer recognizes a special class of linear programming problems with network structure. It uses highly efficient network algorithms on that part of the problem to find a solution from which it then constructs an advanced basis for the rest of your problem. From this advanced basis, ILOG CPLEX then iterates to find a solution to the full problem. Chapter 10, *Solving Network-Flow Problems* explores this optimizer in greater detail.

Barrier Optimizer

The barrier optimizer offers an approach particularly efficient on large, sparse problems (for example, more than 1000 rows or columns, and no more than perhaps a dozen nonzeros per column). The barrier optimizer is sufficiently different in nature from the other optimizers that it is discussed in detail in the next chapter, Chapter 9, *Solving LPs: Barrier Optimizer*.

Sifting Optimizer

Sifting was developed to exploit the characteristics of models with large aspect ratios (that is, a large ratio of the number of columns to the number of rows). In particular, the method is well suited to large aspect ratio models where an optimal solution can be expected to place most variables at their lower bounds. The sifting algorithm can be thought of as an extension to the familiar simplex method. It starts by solving a subproblem (known as the working problem) consisting of all rows but only a small subset of the full set of columns, by assuming an arbitrary value (such as its lower bound) for the solution value of each of the remaining columns. This solution is then used to re-evaluate the reduced costs of the remaining columns. Any columns whose reduced costs violate the optimality criterion become candidates to be added to the working problem for the next major sifting iteration. When no candidates are present, the solution of the working problem is optimal for the full problem, and sifting terminates.

The choice of optimizer to solve the working problem is governed by the `SiftAlg` parameter. You can set this parameter to any of the values accepted by the `LPMethod` parameter, except for `Concurrent` and of course `Sifting` itself. At the default `SiftAlg` setting, ILOG CPLEX chooses the optimizer automatically, typically switching between barrier and primal simplex as the optimization proceeds. It is recommended that you not turn off the barrier crossover step (that is, do not set the parameter `BarCrossAlg` to -1) when you use the sifting optimizer, so that this switching can be carried out as needed.

Concurrent Optimizer

The concurrent optimizer launches distinct optimizers on multiple threads. When the concurrent optimizer is launched on a single-threaded platform, it calls the dual simplex optimizer. In other words, choosing the concurrent optimizer makes sense only on a multiprocessor computer where threads are enabled. For more information about the concurrent optimizer, see Chapter 27, *Parallel Optimizers*, especially *Concurrent Optimizer* on page 540.

Parameter Settings and Optimizer Choice

When you are using parameter settings other than the default, consider the algorithms that these settings will affect. Some parameters, such as the time limit, will affect all the algorithms invoked by the concurrent optimizer. Others, such as the refactoring frequency, will affect both the primal and dual simplex algorithms. And some parameters, such as the primal gradient, dual gradient, or barrier convergence tolerance, affect only a single algorithm.

Tuning LP Performance

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

As a first step in tuning performance, try the different ILOG CPLEX optimizers, as recommended in *Choosing an Optimizer for Your LP Problem* on page 184. The following sections suggest other features of ILOG CPLEX to consider in tuning the performance of your application:

- *Preprocessing* on page 188
- *Starting from an Advanced Basis* on page 190
- *Simplex Parameters* on page 191

Preprocessing

At default settings, ILOG CPLEX preprocesses problems by simplifying constraints, reducing problem size, and eliminating redundancy. Its presolver tries to reduce the size of a problem by making inferences about the nature of any optimal solution to the problem. Its aggregator tries to eliminate variables and rows through substitution. For most models, preprocessing is beneficial to the total solution speed, and ILOG CPLEX reports the model's solution in terms of the user's original formulation, making the exact nature of any reductions immaterial.

Dual Formulation in Presolve

A useful preprocessing feature for performance tuning, one that is not always activated by default, can be to convert the problem to its dual formulation. The nature of the dual formulation is rooted in linear programming theory, beyond the scope of this manual, but for the purposes of this preprocessing feature it is sufficient to think of the roles of the rows and columns of the model's constraint matrix as being switched. Thus the feature is especially applicable to models that have many more rows than columns.

You can direct the preprocessor to form the dual by setting the `PreDual` parameter to `1` (one).

Conversely, to entirely inhibit the dual formulation for the barrier optimizer, you can set the `PreDual` parameter to `-1`. The default, automatic, setting is `0`.

It is worth emphasizing, to those familiar with linear programming theory, that the decision to solve the dual formulation of your model, via this preprocessing parameter, is not the same as the choice between using the dual simplex method or the primal simplex method to perform the optimization. Although these two concepts (dual formulation and dual simplex optimizer) have theoretical foundations in common, it is valid to consider, for example, solving the dual formulation of your model with the dual simplex method; this would not simply result in the same computational path as solving the primal formulation with the primal simplex method. However, with that distinction as background, it may be worth knowing that when CPLEX generates the dual formulation, and a simplex optimizer is to be used, CPLEX will in most cases automatically select the opposite simplex optimizer to the one it would have selected for the primal formulation. Thus, if you set the `PreDual` parameter to 1 (one), and also select LPMethod 1 (which normally invokes the primal simplex optimizer), the dual simplex optimizer will be used in solving the dual formulation. Because solution status and the other results of an optimization are the same regardless of these settings, no additional steps need to be taken by the user to use and interpret the solution; but examination of solution logs might prove confusing if this behavior is not taken into account.

Dependency Checking in Presolve

The ILOG CPLEX preprocessor offers a dependency checker which strengthens problem reduction by detecting redundant constraints. Such reductions are usually most effective

with the barrier optimizer, but these reductions can be applied to all types of problems: LP, QP, QCP, MIP, MIQP, MIQCP. Table 8.3 shows you the possible settings of `DepInd`, the parameter that controls dependency checking, and indicates their effects.

Table 8.3 *Dependency Checking Parameter DepInd or CPX_PARAM_DEPIND*

Setting	Effect
-1	automatic: let CPLEX choose when to use dependency checking
0	turn off dependency checking
1	turn on only at the beginning of preprocessing
2	turn on only at the end of preprocessing
3	turn on at beginning and at end of preprocessing

Final Factor after Presolve

When presolve makes changes to the model prior to optimization, a reverse operation (uncrush) occurs at termination to restore the full model with its solution. With default settings, the simplex optimizers will perform a final basis factorization on the full model before terminating. If you set the `FinalFactor` parameter to Off, the final factorization after uncrushing will be skipped; on large models this can save some time, but computations that require a factorized basis after optimization (for example the computation of the condition number Kappa) may be unavailable depending on the operations presolve performed.

Factorization can easily be performed later by a call to a simplex optimizer with the parameter `AdvInd` set to a value greater than or equal to one.

Memory Use and Presolve

To reduce memory use, presolve may compress the arrays used for storage of the original model. This can make more memory available for the use of the optimizer that the user has called. With default settings, ILOG CPLEX will perform this compression only when using the out-of-core variant of the barrier optimizer (discussed in *Solving LPs: Barrier Optimizer* on page 225). You can explicitly turn this feature on or off by setting the presolve compression parameter `PreCompress` to `-1` for off, or `1` for on; the default of `0` specifies the automatic setting.

Controlling Passes in Preprocessing

In rare instances, a user may wish to specify the number of analysis passes that the presolver or the aggregator makes through the problem. The parameters `PrePass` and `AggInd`, respectively, control these two preprocessing features; the default, `automatic`, setting of `-1` lets ILOG CPLEX determine the number of passes to make, while a setting of `0` directs ILOG CPLEX **not** to use that preprocessing feature, and a positive integer limits the number

of passes to that value. At the automatic setting, ILOG CPLEX applies the aggregator just once when it is solving an LP model; for some problems, it may be worthwhile to increase the `AggInd` setting. The behavior under the `PrePass` default is less easy to predict, but if the output log indicates it is performing excessive analysis you may wish to try a limit of five passes or some other modest value.

Aggregator Fill in Preprocessing

Another parameter, which affects only the aggregator, is `AggFill`. Occasionally the substitutions made by the aggregator will increase matrix density and thus make each iteration too expensive to be advantageous. In such cases, try lowering `AggFill` from its default value of `10`. ILOG CPLEX may make fewer substitutions as a consequence, and the resulting problem will be less dense.

Turning Off Preprocessing

Finally, if for some reason you wish to turn ILOG CPLEX preprocessing entirely off, set the parameter `PreInd` to `0`.

Starting from an Advanced Basis

Another performance improvement to consider, unless you are using the barrier optimizer, is starting from an advanced basis. If you can start a simplex optimizer from an advanced basis, then there is the potential for the optimizer to perform significantly fewer iterations, particularly when your current problem is similar to a problem that you have solved previously. Even when problems are different, starting from an advanced basis may possibly help performance. For example, if your current problem is composed of several smaller problems, an optimal basis from one of the component problems may significantly speed up solution of the other components or even of the full problem.

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

In cases where models are solved in separate application calls, and thus the basis will not be available in memory, you can communicate the final basis from one run to the start of the next by first saving the basis to a file before the end of the first run.

To save the basis to a file:

- When you are using the Component Libraries, use the method `cplex.writeBasis` or the Callable Library routine `CPXmbasewrite`, after the call to the optimizer.
- In the Interactive Optimizer, use the `write` command with the file type `bas`, after optimization.

Then to later read an advanced basis from this file:

- When you are using the Component Libraries, use the method `cplex.readBasis` or the routine `CPXreadcopybase`.
- In the Interactive Optimizer, use the `read` command with the file type `bas`.

Make sure that the advanced start parameter, `AdvInd`, is set to either 1 (its default value) or 2, and not 0 (zero), before calling the optimization routine that is to make use of an advanced basis.

The two nonzero settings for `AdvInd` differ in this way:

- `AdvInd`=1 causes preprocessing to be skipped;
- `AdvInd`=2 invokes preprocessing on both the problem and the advanced basis.

If you anticipate the advanced basis to be a close match for your problem, so that relatively few iterations will be needed, or if you are unsure, then the default setting of 1 is a good choice because it avoids some overhead processing. If you anticipate that the simplex optimizer will require many iterations even with the advanced basis, or if the model is large and preprocessing typically removes much from the model, then the setting of 2 may give you a faster solution by giving you the advantages of preprocessing. However, in such cases, you might also consider **not** using the advanced basis, by setting this parameter to 0 instead, on the grounds that the basis may not be giving you a helpful starting point after all.

Simplex Parameters

After you have chosen the right optimizer and, if appropriate, you have started from an advanced basis, you may want to experiment with different parameter settings to improve performance. This section documents parameters that are most likely to affect performance of the simplex linear optimizers. (The barrier optimizer is different enough from the simplex optimizers that it is discussed elsewhere, in Chapter 9, *Solving LPs: Barrier Optimizer*). The simplex tuning suggestions appear in the following topics:

- *Pricing Algorithm and Gradient Parameters* on page 191
- *Scaling* on page 193
- *Crash* on page 194
- *Memory Management and Problem Growth* on page 194

Pricing Algorithm and Gradient Parameters

The parameters in Table 8.4 determine the pricing algorithms that ILOG CPLEX uses. Consequently, these are the algorithmic parameters most likely to affect simplex linear programming performance. The default setting of these gradient parameters chooses the pricing algorithms that are best for most problems. When you are selecting alternate pricing algorithms, look at these values as guides:

- overall solution time;

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

ILOG CPLEX records those values in the log file as it works. (By default, ILOG CPLEX creates the log file in the directory where it is executing, and it names the log file `cplex.log`. *Managing Log Files* on page 161 tells you how to rename and relocate this log file.)

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

The symbolic names for the acceptable values for these parameters appear in Table 8.4 and Table 8.5. The default value in both cases is `0` (zero).

Table 8.4 *DPriInd Parameter Settings for Dual Simplex Pricing Algorithm*

	Description	Concert	Callable Library
0	determined automatically	`DPriIndAuto`	`CPX_DPRIIND_AUTO`
1	standard dual pricing	`DPriIndFull`	`CPX_DPRIIND_FULL`
2	steepest-edge pricing	`DPriIndSteep`	`CPX_DPRIIND_STEEP`
3	steepest-edge in slack space	`DPriIndFullSteep`	`CPX_DPRIIND_FULLSTEEP`
4	steepest-edge, unit initial norms	`DPriIndSteepQStart`	`CPX_DPRIIND_STEEPQSTART`
5	devex pricing	`DPriIndDevex`	`CPX_DPRIIND_DEVEX`

Table 8.5 *PPriInd Parameter Settings for Primal Simplex Pricing Algorithm*

	Description	Concert	Callable Library
-1	reduced-cost pricing	`PPriIndPartial`	`CPX_PPRIIND_PARTIAL`
0	hybrid reduced-cost and devex	`PPriIndAuto`	`CPX_PPRIIND_AUTO`
1	devex pricing	`PPriIndDevex`	`CPX_PPRIIND_DEVEX`
2	steepest-edge pricing	`PPriIndSteep`	`CPX_PPRIIND_STEEP`
3	steepest-edge, slack initial norms	`PPriIndSteepQStart`	`CPX_PPRIIND_STEEPQSTART`
4	full pricing	`PriIndFull`	`CPX_PPRIIND_FULL`

For the dual simplex pricing parameter, the default value selects steepest-edge pricing. That is, the default (0 or `CPX_DPRIIND_AUTO`) automatically selects 2 or `CPX_DPRIIND_STEEP`.

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

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

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

Scaling

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

Scaling is determined by the parameter `ScaInd`, and may be set as in Table 8.6.

Table 8.6 *ScaIndParameter Settings for Scaling Methods*

ScaInd Value	Meaning
`-1`	no scaling
`0`	equilibration scaling (default)
`1`	aggressive scaling

Refactoring Frequency

ILOG CPLEX dynamically determines the frequency at which the basis of a problem is refactored in order to maximize the speed of iterations. On very large problems, ILOG CPLEX refactors the basis matrix infrequently. Very rarely should you consider

increasing the number of iterations between refactoring. The refactoring interval is controlled by the `ReInv` parameter. The default value of `0` (zero) means ILOG CPLEX will decide dynamically; any positive integer specifies the user's chosen factoring frequency.

Crash

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

In the dual simplex optimizer, the `CraInd` setting determines whether ILOG CPLEX aggressively uses primal variables instead of slack variables while it still tries to preserve as much dual feasibility as possible. If its value is `1` (one), it indicates the default starting basis; if its value is `0` (zero) or `-1`, it indicates an aggressive starting basis. These settings are summarized in Table 8.7.

Table 8.7 *CraInd Parameter Settings for the Dual Simplex Optimizer*

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

In the primal simplex optimizer, the `CraInd` setting determines how ILOG CPLEX uses the coefficients of the objective function to select the starting basis. If its value is `1` (one), ILOG CPLEX uses the coefficients to guide its selection; if its value is `0` (zero), ILOG CPLEX ignores the coefficients; if its value is `-1`, ILOG CPLEX does the opposite of what the coefficients normally suggest. These settings are summarized in Table 8.8.

Table 8.8 *CraInd Parameter Settings for the Primal Simplex Optimizer*

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

Memory Management and Problem Growth

ILOG CPLEX automatically handles memory allocations to accommodate the changing size of a problem object as you modify it. And it manages (using a cache) most changes to prevent inefficiency when the changes will require memory re-allocations. If an application builds a large problem in small increments, you may be able to improve performance by increasing the growth parameters, `RowGrowth`, `ColGrowth`, and `NzGrowth`. In particular, if

you know reasonably accurate upper bounds on the number of rows, columns, and nonzeros, and you are building a large problem in very small pieces with many calls to the problem modification routines, then setting the growth parameters to the known upper bounds will make ILOG CPLEX perform the fewest allocations and may thus improve performance of the problem modification routines. However, overly generous upper bounds may result in excessive memory use.

Diagnosing Performance Problems

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

Lack of Memory

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

Warning Messages

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

Paging Virtual Memory

If you observe paging of memory to disk, then your application is incurring a performance penalty. If you increase available memory in such a case, performance will speed up dramatically.

Refactoring Frequency and Memory Requirements

The ILOG CPLEX primal and dual simplex optimizers refactor the problem basis at a rate determined by the `ReInv` parameter.

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

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

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

Preprocessing and Memory Requirements

By default, ILOG CPLEX automatically preprocesses your problem before optimizing, and this preprocessing requires memory. If memory is extremely tight, consider turning off preprocessing, by setting the `PreInd` parameter to `0`. But doing this foregoes the potential performance benefit of preprocessing, and should be considered only as a last resort. Alternatively, consider turning on presolve compression using the `PreCompress` parameter.

Turning off the `FinalFactor` parameter may help to avoid exhausting memory when such problems occur at the very end of optimization. For an explanation of that parameters, see *Final Factor after Presolve* on page 189.

Numerical Difficulties

ILOG CPLEX is designed to handle the numerical difficulties of linear programming automatically. In this context, numerical difficulties mean such phenomena as:

- repeated occurrence of singularities;
- little or no progress toward reaching the optimal objective function value;
- little or no progress in scaled infeasibility;
- repeated problem perturbations; and
- repeated occurrences of the solution becoming infeasible.

While ILOG CPLEX will usually achieve an optimal solution in spite of these difficulties, you can help it do so more efficiently. This section characterizes situations in which you can help.

Some problems will not be solvable even after you take the measures suggested here. For example, problems can be so poorly conditioned that their optimal solutions are beyond the numerical precision of your computer.

Numerically Sensitive Data

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

Placing large upper bounds (say, in the neighborhood of *1e9* to *1e12*) on individual variables can cause difficulty during Presolve. If you intend for such large bounds to mean "no bound is really in effect" it is better to simply not include such bounds in the first place.

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

A related source of difficulty is the form of rounding when fractions are represented as decimals; expressing $^1/_3$ as .33333333 on a computer that in principle computes values to about 16 digits can introduce an apparent "exact" value that will be treated as given but may not represent what you intend. This difficulty is compounded if similar or related values are represented a little differently elsewhere in the model.

For example, consider the constraint `1/3 x1 + 2/3 x2 = 1`. Under perfect arithmetic, it is equivalent to `x1 + 2 x2 = 3`. However, if you express the fractional version with decimal data values, some truncation is unavoidable. If you happen to include the truncated decimal version of the constraint in the same model with some differently-truncated version or even the exact-integer data version, an unexpected result could easily occur. Consider the following problem formulation:

```
Maximize
 obj: x1 + x2
Subject To
 c1: 0.333333 x1 + 0.666667 x2  = 1
 c2: x1 + 2 x2  = 3
End
```

With default numerical tolerances, this will deliver an optimal solution of `x1=1.0` and `x2=1.0`, giving an objective function value of `2.0`. Now, see what happens when using slightly more accurate data (in terms of the fractional values that are clearly intended to be expressed):

```
Maximize
 obj: x1 + x2
Subject To
 c1: 0.333333333 x1 + 0.666666667 x2  = 1
 c2: x1 + 2 x2  = 3
End
```

The solution to this problem has `x1=3.0` and `x2=0.0`, giving an optimal objective function value of `3.0`, a result qualitatively different from that of the first model. Since this latter result is the same as would be obtained by removing constraint `c1` from the model entirely, this is a more satisfactory result. Moreover, the numerical stability of the optimal basis (as indicated by the condition number, discussed in the next section), is vastly improved.

The result of the extra precision of the input data is a model that is less likely to be sensitive to rounding error or other effects of solving problems on finite-precision computers, or in extreme cases will be more likely to produce an answer in keeping with the intended model. The first example, above, is an instance where the data truncation has fundamentally distorted the problem being solved. Even if the exact-integer data version of the constraint is not present with the decimal version, the truncated decimal version no longer exactly represents the intended meaning and, in conjunction with other constraints in your model, could give unintended answers that are "accurate" in the context of the specific data being fed to the optimizer.

Be particularly wary of data in your model that has been computed (within your program, or transmitted to your program from another via an input file) using single-precision (32 bit) arithmetic. For example in C this would come from using type *float* instead of *double*. Such data will be accurate to only about 8 decimal digits, so that for instance if you print the data you might see values like `0.3333333432674408` instead of `0.3333333333333333`. ILOG CPLEX uses double-precision (64 bit) arithmetic in its computations, and truncated single-precision data carries the risk that it will convey a different meaning than the user intends.

The underlying principle behind all the cautions in this section is that information contained in the data needs to reflect actual meaning or the optimizer may reach unstable solutions or encounter algorithmic difficulties.

Measuring Problem Sensitivity with Basis Condition Number

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

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

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

- In the Interactive Optimizer, use the command `display solution kappa` in order to see the basis condition number of a resident basis matrix.
- In Concert Technology, use the method:

  `IloCplex::getQuality(IloCplex::Kappa)` (C++)

  `IloCplex.getQuality(IloCplex.QualityType.Kappa)` (Java)

  `Cplex.GetQuality(Cplex.QualityType.Kappa)` (.NET)
- In the Callable Library, use the routine `CPXgetdblquality` to access the condition number in the double-precision variable `dvalue`, like this:

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

Repeated Singularities

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

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

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

If ILOG CPLEX encounters repeated singularities in your problem, you may want to try alternative scaling on the problem (rather than simply increasing ILOG CPLEX tolerance for singularities). *Scaling* on page 193 explains how to try alternative scaling.

If alternate scaling does not help, another tactic to try is to increase the Markowitz tolerance. The Markowitz tolerance controls the kinds of pivots permitted. If you set it near its maximum value of `0.99999`, it may make iterations slower but more numerically stable. *Inability to Stay Feasible* on page 200 shows how to change the Markowitz tolerance.

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

Stalling Due to Degeneracy

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

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

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

To start perturbation yourself, set the parameter `PerInd` to `1` instead of its default value of `0`. The perturbation constant, `EpPer`, is usually appropriate at its default value of $1e^{-6}$, but can be set to any value $1e^{-8}$ or larger.

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

Inability to Stay Feasible

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

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

Diagnosing LP Infeasibility

ILOG CPLEX reports statistics about any problem that it optimizes. For infeasible solutions, it reports values that you can analyze to determine where your problem formulation proved

infeasible. In certain situations, you can then alter your problem formulation or change ILOG CPLEX parameters to achieve a satisfactory solution.

- When the ILOG CPLEX *primal* simplex optimizer terminates with an infeasible basic solution, it calculates dual variables and reduced costs relative to the Phase I objective function; that is, relative to the infeasibility function. The Phase I objective function depends on the current basis. Consequently, if you use the primal simplex optimizer with various parameter settings, an infeasible problem will produce different objective values and different solutions.
- In the case of the dual simplex optimizer, termination with a status of infeasibility occurs only during Phase II. Therefore, all solution values are relative to the problem's natural (primal) formulation, including the values related to the objective function, such as the dual variables and reduced costs. As with the primal simplex optimizer, the basis at which the determination of infeasibility is made may not be unique.

ILOG CPLEX provides tools to help you analyze the source of the infeasibility in your model. Those tools include the IIS Finder and FeasOpt:

- The IIS Finder finds a set of irreducibly inconsistent constraints in a model. For more about this this feature, see *Finding a Set of Irreducibly Inconsistent Constraints* on page 204.
- FeasOpt is implemented in the Callable Library by the routine `CPXfeasopt` and in Concert Technology by the method `feasOpt`. For more about this feature, see *More about Infeasibility: FeasOpt* on page 210.

With the help of those tools, you may be able to modify your problem to avoid infeasibility.

The Effect of Preprocessing on Feasibility

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

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

Thus, to aid analysis of an infeasible or unbounded declaration by the preprocessor, a parameter is provided that the user can set, so that the optimization can be rerun to make sure that the results reported by the preprocessor can be interpreted. If a model is declared by

the preprocessor to be infeasible or unbounded and the user believes that it might be infeasible, the parameter `Reduce` can be set to `1` by the user, and the preprocessor will only perform primal reductions. If the preprocessor still finds inconsistency in the model, it will be declared by the preprocessor to be infeasible, instead of infeasible or unbounded. Similarly, setting the parameter to `2` means that if the preprocessor detects unboundedness in the model, it will be declared unambiguously to be unbounded.

To control the types of reductions performed by the presolver, set the `Reduce` parameter to one of the following values:

- 0 = no primal and dual reductions
- 1 = only primal reductions
- 2 = only dual reductions
- 3 = both primal and dual reductions (default)

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

Coping with an Ill-Conditioned Problem or Handling Unscaled Infeasibilities

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

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

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

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

You may also be able to re-optimize the problem successfully after you reduce optimality tolerance, as explained in *Maximum Reduced-Cost Infeasibility* on page 203, or after you reduce feasibility tolerance, as explained in *Maximum Bound Infeasibility: Identifying*

Largest Bound Violation on page 203. When you change these tolerances, ILOG CPLEX may produce a better solution to your problem, but lowering these tolerances sometimes produces erratic behavior or an unstable optimal basis.

Check the basis condition number, as explained in *Measuring Problem Sensitivity with Basis Condition Number* on page 198. If the condition number is fairly low (for example, as little as `1e5` or less), then you can be confident about the solution. If the condition number is high, or if reducing tolerance does not help, then you must revise the model because the current model may be too ill-conditioned to produce a numerically reliable result.

Interpreting Solution Statistics

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

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

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

The following sections discuss these summary statistics in greater detail.

Maximum Bound Infeasibility: Identifying Largest Bound Violation

The maximum bound infeasibility identifies the largest bound violation. This information may help you determine the cause of infeasibility in your problem. If the largest bound violation exceeds the feasibility tolerance of your problem by only a small amount, then you may be able to get a feasible solution to the problem by increasing the feasibility tolerance parameter, `EpRHS`. Its range is between `1e-9` and `0.1`. Its default value is `1e-6`.

Maximum Reduced-Cost Infeasibility

The maximum reduced-cost infeasibility identifies a value for the optimality tolerance that would cause ILOG CPLEX to perform additional iterations. It refers to the infeasibility in the dual slack associated with reduced costs. Whether ILOG CPLEX terminated with an optimal or infeasible solution, if the maximum reduced-cost infeasibility is only slightly

smaller in absolute value than the optimality tolerance, then solving the problem with a smaller optimality tolerance may result in an improvement in the objective function.

To change the optimality tolerance, set the parameter `EpOpt`.

Maximum Row Residual

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

Maximum Dual Residual

The maximum dual residual indicates the numerical accuracy of the reduced costs in the current solution. By construction, in exact arithmetic, the dual residual of a basic solution is always 0 (zero). A nonzero value is thus the effect of roundoff error due to finite-precision arithmetic in the computation of the dual solution vector. Thus, a significant nonzero value indicates ill conditioning.

Maximum Absolute Values: Detecting Ill-Conditioned Problems

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

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

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

If you determine from this analysis that your model is indeed ill-conditioned, then you need to reformulate it. *Coping with an Ill-Conditioned Problem or Handling Unscaled Infeasibilities* on page 202 outlines steps to follow in this situation.

Finding a Set of Irreducibly Inconsistent Constraints

If ILOG CPLEX reports that your problem is infeasible, then you can invoke the ILOG CPLEX infeasibility finder to help you analyze the source of the infeasibility. This diagnostic tool computes a set of infeasible constraints and column bounds that would be

feasible if one of them (a constraint or variable) were removed. Such a set is known as an *irreducibly inconsistent set* (IIS).

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

- the problem has been optimized by the primal or dual simplex optimizer or by the barrier optimizer with crossover, and
- the optimizer has terminated with a declaration of infeasibility.

The parameter `IISInd` controls the method used for determining the IIS. Its default setting `0` (zero) invokes an algorithm that requires minimal computation time but may generate a large set of inconsistent constraints. Its alternative value of `1` (one) may take longer but generates a minimal set of irreducibly inconsistent constraints. Use the Concert Technology method `cplex.writeIIS` or the Callable Library routines `CPXdisplayiis` or `CPXiiswrite` to output the results for review.

Correcting Multiple Infeasibilities

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

Infeasibility Finder in the Interactive Optimizer

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

After that command, invoke the infeasibility finder with the command `display iis`. `ILOG CPLEX` will respond like this:

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

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

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

Example: Writing an IIS-Type File

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

```
CPLEX> write infeas.iis

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

CPLEX> xecute edit infeas.iis

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

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

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

Example: Interpreting a Cumulative Constraint

In the example presented thus far, there were sufficiently few row and column bounds in the IIS to see the cause of infeasibility at a glance. In contrast, other IIS files may contain so many rows and columns that it becomes difficult to see the cause of infeasibility. When an IIS contains many equality constraints and only a few bounds, for example, this phenomenon commonly occurs. In such a situation, the equality constraints transfer the infeasibility from one part of the model to another until eventually no more transfers can occur. Consequently, such an IIS file will also contain a cumulative constraint consisting of

the sum of all the equality rows. This cumulative constraint can direct you toward the cause of infeasibility, as the following sample IIS illustrates:

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

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

```
-x24 -x102 + x103 -x105 +x107 -x115 +x116 + x130 -x131 >= -14434
```

so it is impossible to satisfy the sum of equality rows. Therefore, to correct the infeasibility, you must alter one or more of the bounds, or you must increase one or more of the right-hand side values.

Example: Setting a Time Limit on the Infeasibility Finder

The ILOG CPLEX infeasibility finder will stop when its total runtime exceeds the limit set by the command `set timelimit`. The infeasibility finder works by removing constraints

and bounds that cannot be involved in the IIS, so it can provide partial information about an IIS when it reaches its time limit. The collection of constraints and bounds it offers then do not strictly satisfy the definition of an IIS. However, the collection does contain a true IIS within it, and frequently it provides enough information for you to diagnose the cause of infeasibility in your problem. When it reaches the time limit, ILOG CPLEX output indicates that it has found only a partial IIS. The following example illustrates this idea. It sets a time limit and then invokes the feasibility finder.

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

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

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

Tactics for Interpreting IIS Output

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

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

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

More about Infeasibility: FeasOpt

Previous sections focused on how to diagnose the causes of infeasibility. However, you may want to go beyond diagnosis to perform automatic correction of your model and then proceed with delivering a solution. One approach for doing so is to build your model with explicit slack variables and other modeling constructs, so that an infeasible outcome is never a possibility. Such techniques for formulating a model are beyond the scope of this discussion, but you should consider them if you want the greatest possible flexibility in your application.

An automated approach offered in ILOG CPLEX is known as FeasOpt (for feasible optimization). Depending on the interface you are using, you access FeasOpt in one of the ways listed in Table 8.9. FeasOpt is not available in the Interactive Optimizer.

Table 8.9 *FeasOpt*

API	FeasOpt
Callable Library	`CPXfeasopt`
Concert Technology for C++ users	`IloCplex::feasOpt`
Concert Technology for Java users	`IloCplex.feasOpt`
Concert Technology for C#.NET users	`Cplex.FeasOpt`

FeasOpt accepts an infeasible model and selectively relaxes the bounds and constraints in a way that minimizes a weighted penalty function that you define. In essence, FeasOpt tries to suggest the least change that would achieve feasibility. FeasOpt does not actually modify your model. Instead, it returns a suggested set of bounds and constraint ranges, along with the solution that would result from these relaxations. Your application can report these values directly, or it can apply these new values to your model, or you can run FeasOpt again with different weights perhaps to find a more acceptable relaxation.

In the various Concert Technology APIs, you have a choice of three versions of FeasOpt, specifying that you want to allow changes to the bounds on variables, to the ranges on constraints, or to both. In the Callable Library, only the last of these calling sequences is available. In each of the APIs, there is an additional argument where you specify whether you want merely a *feasible* solution suggested by the bounds and ranges that FeasOpt identifies, or an *optimized* solution that uses these bounds and ranges.

You specify the bounds or ranges that FeasOpt may consider for modification, by assigning positive preference values for each. A negative or zero preference means that the associated

bound or range is **not** to be modified. The weighted penalty function is constructed from these preferences, like this: $\sum v_i / p_i$ where v_i is the violation and p_i is the preference.

Thus, the larger the preference, the more likely it will be that a given bound or range will be modified. However, it is not necessary to specify a unique preference for each bound or range. In fact, it is conventional to use only the values 0 (zero) and 1 (one) except when your knowledge of the problem suggests assigning explicit preferences.

The following examples show you how to use FeasOpt. These fragments of code are written in Concert Technology for C++ users, but the same principles apply to the other APIs as well. The examples begin with a model similar to one that you have seen repeatedly in this manual.

```
IloEnv   env;
IloModel model(env);
IloNumVarArray x(env);
IloRangeArray con(env);
IloNumArray vals(env);

x.add(IloNumVar(env, 0.0, 40.0));
x.add(IloNumVar(env));
x.add(IloNumVar(env));

model.add(IloMaximize(env, x[0] + 2 * x[1] + 3 * x[2]));
con.add( - x[0] +     x[1] + x[2] <=  20);
con.add(   x[0] - 3 * x[1] + x[2] <=  30);
con.add(   x[0] +     x[1] + x[2] >= 150);
model.add(con);
```

If you extract that model and solve it, by means of the following lines, you find that it is infeasible.

```
IloCplex cplex(model);
cplex.solve();
if ( cplex.getStatus() == IloAlgorithm::Infeasible ||
     cplex.getStatus() == IloAlgorithm::InfeasibleOrUnbounded ) {
     env.out() << endl << "*** Model is infeasible ***" << endl << endl;
```

Now the following lines invoke FeasOpt to locate a feasible solution:

```
// begin feasOpt analysis

cplex.setOut(env.getNullStream());
IloNumArray lb(env);
IloNumArray ub(env);

// first feasOpt call

env.out() << endl << "*** First feasOpt call ***" << endl;
env.out() << "*** Consider all constraints ***" << endl;
int rows = con.getSize();
lb.add(rows, 1.0);
ub.add(rows, 1.0);

if ( cplex.feasOpt(con, lb, ub, IloTrue) ) {
    env.out() << endl;
    env.out() << "*** Suggested lower range = " << lb << endl;
    env.out() << "*** Suggested upper range = " << ub << endl;
    env.out() << "*** Feasible objective value would be = "
                              << cplex.getObjValue() << endl;
    env.out() << "Solution status    = " << cplex.getStatus() << endl;
    env.out() << "Solution obj value = " << cplex.getObjValue() << endl
           env.out() << "Values             = " << vals << endl;
           env.out() << endl;
    }
    else {
        env.out() << "*** Could not repair the infeasibility" << endl;
        throw (-1);
      }
```

The code first turns off logging to the screen by the optimizers, simply to avoid unnecessary output. It then allocates arrays `lb` and `ub`, to contain the preferences as input, and to receive the modified constraint ranges as output. The preference is set to 1.0 for all three constraints in both directions to indicate that any change to a constraint range will be permissible.

Then FeasOpt is called, with the setting of `IloTrue` for the argument that specifies whether to return an optimized solution. If the FeasOpt call succeeds, then several lines of output give the results. Here is the output:

```
*** First feasOpt call ***
*** Consider all constraints ***

*** Suggested lower range = [-INF, -INF, 150]
*** Suggested upper range = [70, 30, INF]
*** Feasible objective value would be = 340
Solution status    = Infeasible
Solution obj value = 340
Values             = [40, 30, 80]
```

There are several items of note in this output. First, you see that FeasOpt recommends only the first constraint to be modified, namely, by changing the constraint from its original form:

```
- x[0] +     x[1] + x[2] <=  20
```

to the following:

```
- x[0] +     x[1] + x[2] <=  70
```

The solution values of `[40, 30, 80]` would be feasible in the modified version of the constraint, but not in the original version. This situation is reflected by the fact that the solution status has not changed from its value of `Infeasible`. In other words, this change to the right-hand side (RHS) of the constraint is only a suggestion from FeasOpt; the model itself has not changed, and the proposed solution is still infeasible in it.

To get a more concrete idea, assume that this constraint represents a limit on a supply, and assume further that increasing the supply to 70 is not practical. Now rerun FeasOpt, not allowing this constraint to be modified, like this:

```
// second feasOpt call

env.out() << endl << "*** Second feasOpt call ***" << endl;
env.out() << "*** Consider all but first constraint ***" << endl;

// re-set all indicator variables because they now contain values
lb[0]=ub[0]=0.0;
lb[1]=ub[1]=1.0;
lb[2]=ub[2]=1.0;

if ( cplex.feasOpt(con, lb, ub, IloTrue) ) {
     env.out() << endl;
     env.out() << "*** Suggested lower range = " << lb << endl;
     env.out() << "*** Suggested upper range = " << ub << endl;
     env.out() << "*** Feasible objective value would be = "
                                  << cplex.getObjValue() << endl;
     env.out() << "Solution status    = " << cplex.getStatus() << endl;
     env.out() << "Solution obj value = " << cplex.getObjValue() << endl;
     cplex.getValues(vals, x);
     env.out() << "Values             = " << vals << endl;
     env.out() << endl;
    }
    else {
         env.out() << "*** Could not repair the infeasibility" << endl;
         throw (-1);
}
```

In those lines, preferences about the lower and upper bounds, in the arrays `lb` and `ub`, were set because they contained the results from the first FeasOpt call. Also those lines disallow any changes to the first constraint by setting `lb[0]=ub[0]=0.0`. FeasOpt runs again, and here are the results of this second run:

```
*** Second feasOpt call ***
*** Consider all but first constraint ***
*** Suggested lower range = [-INF, -INF, 100]
*** Suggested upper range = [20, 30, INF]
*** Feasible objective value would be = 202.5
Solution status    = Infeasible
Solution obj value = 202.5
Values             = [40, 17.5, 42.5]
```

Notice that the projected maximal objective value is quite different from the first time, as are the optimal values of the three variables. This solution was completely unaffected by the previous call to FeasOpt. This solution also is infeasible with respect to the original model, as you would expect. (If it had been feasible, you would not have needed FeasOpt in the first place.)

That second call changed the range of a constraint. Now consider changes to the bounds.

```
// third feasOpt call

env.out() << endl << "*** Third feasOpt call ***" << endl;
env.out() << "*** Consider all bounds ***" << endl;

// re-use indicator variables - they happen to be right dimension
lb[0]=ub[0]=1.0;
lb[1]=ub[1]=1.0;
lb[2]=ub[2]=1.0;

if ( cplex.feasOpt(x, lb, ub, IloTrue) ) {
    env.out() << endl;
    env.out() << "*** Suggested lower bounds = " << lb << endl;
    env.out() << "*** Suggested upper bounds = " << ub << endl;
    env.out() << "*** Feasible objective value would be = "
                                  << cplex.getObjValue() << endl;
    env.out() << "Solution status    = " << cplex.getStatus() << endl;
    env.out() << "Solution obj value = " << cplex.getObjValue()<< endl;
    cplex.getValues(vals, x);
    env.out() << "Values             = " << vals << endl;
    env.out() << endl;
   }
   else {
        env.out() << "*** Could not repair the infeasibility" << endl;
        throw (-1);
}
```

In those lines, all six bounds (lower and upper bounds of three variables) are considered for possible modification because a preference of 1.0 is set for each of them. Here is the result:

```
*** Third feasOpt call ***
*** Consider all bounds ***

*** Suggested lower bounds = [0, 0, 0]
*** Suggested upper bounds = [65, 1e+20, 1e+20]
*** Feasible objective value would be = 290
Solution status    = Infeasible
Solution obj value = 290
Values             = [65, 30, 55]
```

Those results suggest modifying only one bound, the upper bound on the first variable. (The value 1e+20 serves as Infinity on the variables having no upper bound.) And just as you might expect, the solution value for that first variable is exactly at its upper bound; there is no incentive in the weighted penalty function to set the bound any higher than it has to be to achieve feasibility.

Now assume for some reason it is undesirable to let this variable have its bound modified. The final call to FeasOpt changes the preference to achieve this effect, like this:

```
// fourth feasOpt call

env.out() << endl << "*** Fourth feasOpt call ***" << endl;
env.out() << "*** Consider all bounds except first ***" << endl;
lb[0]=ub[0]=0.0;
lb[1]=ub[1]=1.0;
lb[2]=ub[2]=1.0;

if ( cplex.feasOpt(x, lb, ub, IloTrue) ) {
    env.out() << endl;
    env.out() << "*** Suggested lower bounds = " << lb << endl;
    env.out() << "*** Suggested upper bounds = " << ub << endl;
    env.out() << "*** Feasible objective value would be = "
                                    << cplex.getObjValue() << endl;
    env.out() << "Solution status    = " << cplex.getStatus() << endl;
    env.out() << "Solution obj value = " << cplex.getObjValue() << endl;
    cplex.getValues(vals, x);
    env.out() << "Values             = " << vals << endl;
    env.out() << endl;
   }
    else {
        env.out() << "*** Could not repair the infeasibility" << endl;
        throw (-1);
}
```

Then after the fourth call of FeasOpt, the output to the screen looks like this:

```
*** Fourth feasOpt call ***
*** Consider all bounds except first ***
*** Could not repair the infeasibility
Unknown exception caught
```

This is a correct outcome, and a more nearly complete application should catch this exception and handle it appropriately. FeasOpt is telling the user here that no modification to the model is possible under this set of preferences: only the bounds on the last two variables are permitted to change according to the preferences expressed by the user, and they are already at `[0,+inf]`, so the upper bound can not increase, and no negative value for the lower bounds would ever improve the feasibility of this model. Not every infeasibility can be repaired, and an application calling FeasOpt will usually need to take this possibility into account.

Example: Using a Starting Basis in an LP Problem

Here is an approach mentioned in the section *Tuning LP Performance*, which is to start with an advanced basis. The following small example demonstrates an approach to setting a starting basis by hand.

Example ilolpex6.cpp

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

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

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

The call to `cplex.solve` optimizes the problem, and the subsequent print of the iteration count demonstrates that the advanced basis took effect. In fact, this basis is immediately determined to be the optimal one, resulting in zero iterations being performed, in contrast to the behavior seen in the example program `ilolpex1.cpp` where the same model is solved without the use of an advanced basis.

Complete Program

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

```
// ------------------------------------------------------------ -*- C++ -*-
// File: examples/src/ilolpex6.cpp
// Version 9.0
// ---------------------------------------------------------------------------
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ---------------------------------------------------------------------------
//
// ilolpex6.cpp - Illustrates that optimal basis can be copied and
//                used to start an optimization.

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

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

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

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

      IloCplex cplex(model);

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

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

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

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

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

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

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

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

   model.add(obj);
   model.add(c);
}  // END populatebycolumn
```

Example lpex6.c

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

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

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

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

Complete Program

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

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/lpex6.c                                              */
/*  Version 9.0                                                             */
```

```
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2003 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* lpex6.c - Illustrates that optimal basis can be copied and
             used to start an optimization. */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

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

#include <string.h>

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

static int
   populatebycolumn  (CPXENVptr env, CPXLPptr lp);



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

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

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

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

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

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

/* Initialize the CPLEX environment */

env = CPXopenCPLEX (&status);

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

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

/* Turn on output to the screen */

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

/* Create the problem. */

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

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

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

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

```
status = populatebycolumn (env, lp);

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

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

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

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

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

/* Now copy the basis */

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

/* Optimize the problem and obtain solution. */

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

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


/* Write the output to the screen. */

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

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

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

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

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

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

TERMINATE:

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

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

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

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

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

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

   return (status);

}  /* END main */
```

```
/* This function builds by column the linear program:

      Maximize
       obj: x1 + 2 x2 + 3 x3
      Subject To
       c1: - x1 + x2 + x3 <= 20
       c2: x1 - 3 x2 + x3 <= 30
      Bounds
       0 <= x1 <= 40
      End
 */

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

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

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

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

   rowname[0] = "c1";
   sense[0]   = 'L';
   rhs[0]     = 20.0;

   rowname[1] = "c2";
   sense[1]   = 'L';
   rhs[1]     = 30.0;

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

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

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

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

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

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

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

   colname[0] = "x1"; colname[1] = "x2";      colname[2] = "x3";

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

TERMINATE:

   return (status);

}  /* END populatebycolumn */
```

C H A P T E R

9

Solving LPs: Barrier Optimizer

This chapter tells you more about solving linear programming problems by means of the ILOG CPLEX *Barrier Optimizer*. It includes sections about:

- *Identifying LPs for Barrier Optimization* on page 226
- *Using the Barrier Optimizer* on page 228
- *Special Options* on page 229
- *Controlling Crossover* on page 229
- *Using VEC File Format* on page 230
- *Interpreting the Barrier Log File* on page 230
- *Understanding Solution Quality from the Barrier LP Optimizer* on page 233
- *Tuning Barrier Optimizer Performance* on page 235
- *Overcoming Numerical Difficulties* on page 239
- *Diagnosing Barrier Optimizer Infeasibility* on page 242

Identifying LPs for Barrier Optimization

The ILOG CPLEX Barrier Optimizer is well suited to large, sparse problems. An alternative to the simplex optimizers, it exploits a primal-dual logarithmic barrier algorithm to generate a sequence of strictly positive primal and dual solutions to a problem. As with the simplex optimizers, it is not really necessary to understand the internal workings of barrier in order to obtain its performance benefits. However, for the interested reader, here is an outline of how it works.

ILOG CPLEX finds the primal solutions, conventionally denoted *(x, s)*, from the primal formulation:

Minimize $\boldsymbol{c}^{\mathrm{T}}x$

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

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

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

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

Maximize $b^{\mathrm{T}}y - u^{\mathrm{T}}w + l^{\mathrm{T}}z$

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

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

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

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

On each iteration of the barrier optimizer, ILOG CPLEX computes a matrix based on $\boldsymbol{A}\boldsymbol{A}^{\mathrm{T}}$ and then computes a *Cholesky factor* of it. This factored matrix has the same number of nonzeros on each iteration. The number of nonzeros in this matrix is influenced by the barrier *ordering* parameter.

The ILOG CPLEX Barrier Optimizer is appropriate and often advantageous for large problems, for example, those with more than 1000 rows or columns. It is effective on

problems with staircase structures or banded structures in the constraint matrix. It is also effective on problems with a small number of nonzeros per column.

Its performance is most dependent on these characteristics:

- the number of floating-point operations required to compute the Cholesky factor;
- the presence of dense columns, that is, columns with a relatively high number of nonzero entries.

To decide whether to use the barrier optimizer on a given problem, you should look at both these characteristics. (How to check those characteristics is explained later in this chapter in *Cholesky Factor in the Log File* on page 232, and *Nonzeros in Lower Triangle of AA^T in the Log File* on page 231).

Barrier Simplex Crossover

Since many users prefer basic solutions because they can be used to restart simplex optimization, the ILOG CPLEX Barrier Optimizer includes *basis crossover algorithms*. By default, the barrier optimizer automatically invokes a primal crossover when the barrier algorithm terminates (unless termination occurs abnormally because of insufficient memory or numerical difficulties). Optionally, you can also execute barrier optimization with a dual crossover or with no crossover at all. The section *Controlling Crossover* on page 229 explains how to control crossover in the Interactive Optimizer.

Differences between Barrier and Simplex Optimizers

The barrier optimizer and the simplex optimizers (primal and dual) are fundamentally different approaches to solving linear programming problems. The key differences between them have these implications:

- Simplex and barrier optimizers differ with respect to the *nature of solutions*. Barrier solutions tend to be midface solutions. In cases where multiple optima exist, barrier solutions tend to place the variables at values between their bounds, whereas in basic solutions from a simplex technique, the values of the variables are more likely to be at either their upper or their lower bound. While objective values will be the same, the nature of the solutions can be very different.

  The fact that barrier without crossover does not produce a basic solution has other consequences. Without a basis, you will not be able to optimize the same or similar problems repeatedly using advanced start information. You will also not be able to obtain range information for performing sensitivity analysis.

- Simplex and barrier optimizers have different *numerical properties*, sensitivity, and behavior. For example, the barrier optimizer is sensitive to the presence of unbounded optimal faces, whereas the simplex optimizers are not. As a result, problems that are numerically difficult for one method may be easier to solve by the other.

- Simplex and barrier optimizers have different *memory requirements*. Depending on the size of the Cholesky factor, the barrier optimizer can require significantly more memory than the simplex optimizers.
- Simplex and barrier optimizers work well on different *types of problems*. The barrier optimizer works well on problems where the AA^T remains sparse. Also, highly degenerate problems that pose difficulties for the primal or dual simplex optimizers may be solved quickly by the barrier optimizer. In contrast, the simplex optimizers will probably perform better on problems where the AA^T and the resulting Cholesky factor are relatively dense, though it is sometimes difficult to predict from the dimensions of the model when this will be the case.

Using the Barrier Optimizer

As you have read in *Identifying LPs for Barrier Optimization* on page 226, the ILOG CPLEX Barrier Optimizer finds primal and dual solutions from the primal and dual formulations of a model, but you do not have to reformulate the problem yourself. The ILOG CPLEX Barrier Optimizer automatically creates the primal and dual formulations of the problem for you after you enter or read in the problem.

Specify that you want to use the barrier optimizer by setting the parameter `LPMethod` to one of the values in Table 9.1.

Table 9.1 *Settings of LPMethod Invoke the Barrier Optimizer*

Setting	Context
`IloCplex::Barrier`	Concert Technology for C++ users
`IloCplex.Algorithm.Barrier`	Concert Technology for Java users
`Cplex.Algorithm.Barrier`	Concert Technology for C#.NET users
`CPX_ALG_BARRIER`	Callable Library
4	Interactive Optimizer

And then you call the solution routine just as for any other ILOG CPLEX optimizer, as you see in Table 9.2.

Table 9.2 *Calling the Barrier Optimizer*

Call	Context
`cplex.solve`	Concert Technology for C++ users
`cplex.solve`	Concert Technology for Java users
`Cplex.Solve`	Concert Technology for C#.NET users
`CPXlpopt`	Callable Library
`optimize`	Interactive Optimizer

Special Options

In addition to the parameters available for other ILOG CPLEX LP optimizers, there are also parameters to control the ILOG CPLEX Barrier Optimizer. In the Interactive Optimizer, to see a list of the parameters specific to the ILOG CPLEX Barrier Optimizer, use the command `set barrier`.

Controlling Crossover

The nature of the crossover step that follows barrier is controlled by the parameter `BarCrossAlg`. Under the default `Automatic` setting, an appropriate crossover step will be invoked. Possible settings for the parameter appear in Table 9.3.

Table 9.3 *BarCrossAlg Parameter Settings*

BarCrossAlg Values	Meaning
-1	no crossover
0	automatic (default)
1	primal crossover
2	dual crossover

Using VEC File Format

When you use the ILOG CPLEX Barrier Optimizer with no crossover, you can save the primal and dual variable values and their associated reduced cost and dual values in a VEC-format file. You can then read that VEC file into ILOG CPLEX before you initiate a crossover at a later time. After you read a VEC file into ILOG CPLEX, all three optimizers—primal simplex, dual simplex, and barrier simplex—automatically invoke crossover.

Even if you have set the advanced basis indicator to `no` (meaning that you do not intend to start from an advanced basis), ILOG CPLEX automatically resets the indicator to `yes` when it reads a VEC file. If you turn off the advanced basis indicator after reading a VEC file, then the simplex optimizers will honor this setting and will not initiate crossover.

Interpreting the Barrier Log File

Like the ILOG CPLEX simplex optimizers, the barrier optimizer records information about its progress in a log file as it works. Some users find it helpful to keep a new log file for each session. By default, ILOG CPLEX records information in a file named `cplex.log`. In the:

- Interactive Optimizer, use the command `set logfile filename` to change the name of the log file.
- Callable Library, use the routine `CPXsetlogfile` with arguments to indicate the log file.

You can control the level of information that ILOG CPLEX records about barrier optimization by setting the `BarDisplay` parameter. Those settings appear in Table 9.4.

Table 9.4 *BarDisplay Parameter Settings*

BarDisplay Values	Meaning
0	no display
1	display normal information (default)
2	display detailed (diagnostic) output

Here is an example of a log file for a barrier optimization (without crossover):

```
Tried aggregator 1 time.
LP Presolve eliminated 9 rows and 11 columns.
Aggregator did 6 substitutions.
Reduced LP has 12 rows, 15 columns, and 38 nonzeros.
Presolve time =    0.00 sec.
Number of nonzeros in lower triangle of A*A' = 26
Using Approximate Minimum Degree ordering
Total time for automatic ordering = 0.00 sec.
Summary statistics for Cholesky factor:
  Rows in Factor            = 12
  Integer space required    = 12
  Total non-zeros in factor = 78
  Total FP ops to factor    = 650
 Itn      Primal Obj        Dual Obj  Prim Inf Upper Inf  Dual Inf
   0  -1.3177911e+01  -1.2600000e+03  6.55e+02  0.00e+00  3.92e+01
   1  -4.8683118e+01  -5.4058675e+02  3.91e+01  0.00e+00  1.18e+01
   2  -1.6008142e+02  -3.5969226e+02  1.35e-13  7.11e-15  5.81e+00
   3  -3.5186681e+02  -6.1738305e+02  1.59e-10  1.78e-15  5.16e-01
   4  -4.5808732e+02  -4.7450513e+02  5.08e-12  1.95e-14  4.62e-02
   5  -4.6435693e+02  -4.6531819e+02  1.66e-12  1.27e-14  1.59e-03
   6  -4.6473085e+02  -4.6476678e+02  5.53e-11  2.17e-14  2.43e-15
   7  -4.6475237e+02  -4.6475361e+02  5.59e-13  2.99e-14  2.19e-15
   8  -4.6475312e+02  -4.6475316e+02  1.73e-13  1.55e-14  1.17e-15
   9  -4.6475314e+02  -4.6475314e+02  1.45e-13  2.81e-14  2.17e-15

Barrier - Optimal:  Objective =   -4.6475314194e+02
Solution time =    0.01 sec.  Iterations = 9
```

Preprocessing in the Log File

The opening lines of that log file record information about *preprocessing* by the ILOG CPLEX presolver and aggregator. After those preprocessing statistics, the next line records the number of *nonzeros in the lower triangle* of a particular matrix, $\boldsymbol{AA^T}$, denoted `A*A'` in the log file.

Nonzeros in Lower Triangle of $\boldsymbol{AA}^{\mathrm{T}}$ in the Log File

The number of nonzeros in the lower triangle of $\boldsymbol{AA}^{\mathrm{T}}$ gives an early indication of how long each barrier iteration will take. The larger this number, the more time each barrier iteration requires. If this number is close to 50% of the square of the number of rows, then the problem may contain dense columns that are not being detected. In that case, examine the histogram of column counts; then consider setting the barrier column-nonzeros parameter to a value that enables ILOG CPLEX to treat more columns as being dense.

Ordering-Algorithm Time in the Log File

After the number of nonzeros in the lower triangle of $\boldsymbol{AA}^{\mathrm{T}}$, ILOG CPLEX records the time required by the ordering algorithm. (The ILOG CPLEX Barrier Optimizer offers you a

choice of several ordering algorithms, explained in *Choosing an Ordering Algorithm* on page 238.) This section in the log file indicates which ordering algorithm the default `Automatic` setting chose.

Cholesky Factor in the Log File

After the time required by the ordering algorithm, ILOG CPLEX records information about the *Cholesky factor*. ILOG CPLEX computes this matrix on each iteration. The *number of rows* in the Cholesky factor represents the number after preprocessing. The next line of information about the Cholesky factor—integer space required—indicates the amount of memory needed to store the *sparsity pattern* of the factored matrix. If this number is low, then the factor can be computed more quickly than when the number is high.

Information about the Cholesky factor continues with the number of nonzeros in the factored matrix. The difference between this number and the number of nonzeros in $\boldsymbol{AA}^{\mathrm{T}}$ indicates the *fill-level* of the Cholesky factor.

The final line of information indicates how many floating-point operations are required to compute the Cholesky factor. This number is the best predictor of the time that will be required to perform each iteration of the barrier method.

Iteration Progress in the Log File

After the information about the Cholesky factor, the log file records progress at each iteration. It records both *primal* and *dual objectives* (as `Primal Obj` and `Dual Obj`) per iteration.

It also records absolute infeasibilities per iteration. Internally, the ILOG CPLEX Barrier Optimizer treats inequality constraints as equality constraints with added slack and surplus variables. Consequently, primal constraints in a problem are written as $\boldsymbol{A}x = b$ and $x + s = u$, and the dual constraints are written as $\boldsymbol{A}^{\mathrm{T}}y + z - w = c$. As a result, in the log file, the infeasibilities represent *norms*, as summarized in Table 9.5.

Table 9.5 *Infeasibilities and Norms in the Log File of a Barrier Optimization*

Infeasibility	In log file	Norm
primal	`Prim Inf`	|b - **A**x|
upper	`Upper Inf`	|u - (x + s)|
dual	`Dual Inf`	|c - y**A** - z + w|

If solution values are large in absolute value, then the infeasibilities may appear inordinately large because they are recorded in the log file in absolute terms. The optimizer uses *relative* infeasibilities as termination criteria.

Infeasibility Ratio in the Log File

If you are using one of the barrier infeasibility algorithms available in the ILOG CPLEX Barrier Optimizer (that is, if you have set `BarAlg` to either 1 or 2, as discussed later in this chapter), then ILOG CPLEX records an additional column of output titled `Inf Ratio`, the infeasibility ratio. This ratio, always positive, is a measure of progress for that particular algorithm. In a problem with an optimal solution, you will see this ratio increase to a large number. In contrast, in a problem that is infeasible or unbounded, this ratio will decrease to a very small number.

Understanding Solution Quality from the Barrier LP Optimizer

When ILOG CPLEX successfully solves a problem with the ILOG CPLEX Barrier Optimizer, it reports the optimal objective value and solution time in a log file, as it does for other LP optimizers.

Because barrier solutions (prior to crossover) are not basic solutions, certain solution statistics associated with basic solutions are not available for a strictly barrier solution. For example, reduced costs and dual values are available for strictly barrier LP solutions, but range information about them is not.

To help you evaluate the quality of a barrier solution more readily, ILOG CPLEX offers a special display of information about barrier solution quality. To display this information in the Interactive Optimizer, use the command `display solution quality` after optimization. When using the Component Libraries, use the method `cplex.getQuality` or use the routines `CPXgetintquality` for integer information and `CPXgetdblquality` for double-valued information.

Table 9.6 *Barrier Solution Quality Display*

Item	Meaning
primal objective	primal objective value $c^{\mathrm{T}}x$
dual objective	dual objective value $b^{\mathrm{T}}y - u^{\mathrm{T}}w + l^{\mathrm{T}}z$
duality gap	difference between primal and dual objectives
complementarity	sum of column and row complementarity
column complementarity (total)	sum of $\lvert(x_j - l_j)\bullet z_j\rvert + \lvert(u_j - x_j)\bullet w_j\rvert$

Table 9.6 *Barrier Solution Quality Display (Continued)*

Item	Meaning
column complementarity (max)	maximum of $\|(x_j - l_j) \bullet z_j\|$ and $\|(u_j - x_j) \bullet w_j\|$ over all variables
row complementarity (total)	sum of $\|slack_i \bullet y_i\|$
row complementarity (max)	maximum of $\|slack_i \bullet y_i\|$
primal norm |x| (total)	sum of absolute values of all primal variables
primal norm |x| (max)	maximum of absolute values of all primal variables
dual norm |rc| (total)	sum of absolute values of all reduced costs
dual norm |rc| (max)	maximum of absolute values of all reduced costs
primal error (Ax = b) (total, max)	total and maximum error in satisfying primal equality constraints
dual error (A'`pi` + `rc` = c) (total, max)	total and maximum error in satisfying dual equality constraints
primal x bound error (total, max)	total and maximum error in satisfying primal lower and upper bound constraints
primal slack bound error (total, max)	total and maximum violation in slack variables
dual pi bound error (total, max)	total and maximum violation with respect to zero of dual variables on inequality rows
dual rc bound error (total, max)	total and maximum violation with respect to zero of reduced costs
primal normalized error (Ax = b) (max)	accuracy of primal constraints
dual normalized error (A'pi + rc = c) (max)	accuracy of dual constraints

Table 9.6 lists the items ILOG CPLEX displays and explains their meaning. In the solution quality display, the term `pi` refers to dual solution values, that is, the y values in the conventional barrier problem-formulation. The term `rc` refers to reduced cost, that is, the difference z - w in the conventional barrier problem-formulation. Other terms are best understood in the context of primal and dual LP formulations.

Normalized errors, for example, represent the accuracy of satisfying the constraints while considering the quantities used to compute $\boldsymbol{A}x$ on each row and $y^{\mathrm{T}}\boldsymbol{A}$ on each column. In the

primal case, for each row, consider the nonzero coefficients and the x_j values used to compute $\boldsymbol{A}x$. If these numbers are large in absolute value, then it is acceptable to have a larger absolute error in the primal constraint.

Similar reasoning applies to the dual constraint.

If ILOG CPLEX returned an optimal solution, but the primal error seems high to you, the primal *normalized* error should be low, since it takes into account the scaling of the problem and solution.

After a simplex optimization—whether primal, dual, or network—or after a crossover, the display command will display information related to the quality of the *simplex* solution.

Tuning Barrier Optimizer Performance

Naturally, the default parameter settings for the ILOG CPLEX Barrier Optimizer work best on most problems. However, you can tune several algorithmic parameters to improve performance or to overcome numerical difficulties. The following sections document these parameters:

- *Out-of-Core Barrier: Letting the Optimizer Use Disk for Storage* on page 236;
- *Preprocessing* on page 236;
- *Detecting and Eliminating Dense Columns* on page 237;
- *Choosing an Ordering Algorithm* on page 238;
- *Using a Starting-Point Heuristic* on page 238.

In addition, several parameters set termination criteria. With them, you control when ILOG CPLEX stops optimization.

You can also control convergence tolerance—another factor that influences performance. Convergence tolerance determines how nearly optimal a solution ILOG CPLEX must find: tight convergence tolerance means ILOG CPLEX must keep working until it finds a solution very close to the optimal one; loose tolerance means ILOG CPLEX can return a solution within a greater range of the optimal one and thus stop calculating sooner.

Performance of the ILOG CPLEX Barrier Optimizer is most highly dependent on the number of floating-point operations required to compute the Cholesky factor at each iteration. When you adjust barrier parameters, always check their impact on this number. At default output settings, this number is reported at the beginning of each barrier optimization in the log file, as explained in *Cholesky Factor in the Log File* on page 232.

Another important performance issue is the presence of *dense columns*. A dense column means that a given variable appears in a relatively large number of rows. You can check column density as suggested in *Nonzeros in Lower Triangle of AA^T in the Log File* on

page 231. *Detecting and Eliminating Dense Columns* on page 237 also says more about column density.

In adjusting parameters, you may need to experiment to find beneficial settings because the precise effect of parametric changes will depend on the nature of your LP problem as well as your platform (hardware, operating system, compiler, etc.). Once you have found satisfactory parametric settings, keep them in a parameter specification file for re-use, as explained in the topic *Saving a Parameter Specification File* in the reference manual *ILOG CPLEX File Formats*.

Out-of-Core Barrier: Letting the Optimizer Use Disk for Storage

Under default settings, the ILOG CPLEX Barrier Optimizer will do all of its work using central memory (variously referred to also as RAM, core, or physical memory). For models too large to solve in the central memory on your computer, or in cases where it is simply not desired to use this much memory, it is possible to instruct the barrier optimizer to use disk for part of the working storage it needs, specifically the Cholesky factorization. Since disk is slower than central memory, there may be some lost performance by making this choice on models that could be solved entirely in central memory, but the out-of-core feature in the ILOG CPLEX Barrier Optimizer is designed to make this as efficient as possible. It generally will be far more effective than relying on the operating system's virtual memory (swap space). To activate out-of-core Barrier, set the parameter `BarOOC` to `1` instead of its default value of `0`.

Note that out-of-core barrier uses some working memory to store the portion of the factor on which it is currently performing computation. You can improve performance by allocating more working memory, using the `WorkMem` parameter. More working memory allows the algorithm to transfer less factor data to and from disk. In fact, the factor matrix will not be written to disk at all if its size does not exceed the value of `WorkMem`. The default for this parameter is `128`, meaning 128 megabytes.

When Barrier is being run out-of-core, the location of disk storage is controlled by the Working Directory parameter, `WorkDir`. For example, to use the directory `/tmp/mywork`, set `WorkDir` to the string `'/tmp/mywork'`. `WorkDir` should be specified as the name of a directory that already exists, and ILOG CPLEX will create its working directory under that.

Preprocessing

For best performance of the ILOG CPLEX Barrier Optimizer, preprocessing should almost always be on. That is, use the default setting where the presolver and aggregator are active. While they may use more memory, they also reduce the problem, and problem reduction is crucial to barrier optimizer performance. In fact, reduction is so important that even when you turn off preprocessing, ILOG CPLEX still applies minimal presolving before barrier optimization.

For problems that contain linearly dependent rows, it is a good idea to turn on the *preprocessing dependency parameter*. (By default, it is off.) This dependency checker may add some preprocessing time, but it can detect and remove linearly dependent rows to improve overall performance. Table 9.7 shows you the possible settings of `DepInd`, the parameter that controls dependency checking, and indicates their effects.

Table 9.7 *Dependency Checking Parameter DepInd or CPX_PARAM_DEPIND*

Setting	Effect
-1	automatic: let CPLEX choose when to use dependency checking
0	turn off dependency checking
1	turn on only at the beginning of preprocessing
2	turn on only at the end of preprocessing
3	turn on at beginning and at end of preprocessing

These reductions can be applied to all types of problems: LP, QP, QCP, MIP, MIQP, MIQCP.

Detecting and Eliminating Dense Columns

Dense columns can significantly degrade barrier optimizer performance. A dense column is one in which a given variable appears in many rows. The interactive optimizer contains a display feature that shows a histogram of the number of nonzeros in the columns of your model, `display problem histogram c`.

In fact, when a few dense columns are present in a problem, it is often effective to reformulate the problem to remove those dense columns from the model.

Otherwise, you can control whether ILOG CPLEX perceives columns as dense by setting the column nonzeros parameter. At its default setting, ILOG CPLEX calculates an appropriate value for this parameter automatically. However, if your problem contains one (or a few) dense columns that remain undetected at the default setting, you can adjust this parameter yourself to help ILOG CPLEX detect it (or them). For example, in a large problem in which one column contains forty entries while the other columns contain less than five entries, you may benefit by setting the column nonzeros parameter to 30. This setting allows ILOG CPLEX to recognize that column as dense and thus invoke techniques to handle it.

To set the dense column threshold, set the parameter `BarColNz` to a positive integer. The default value of `0` means that ILOG CPLEX will determine the threshold.

Choosing an Ordering Algorithm

ILOG CPLEX offers several different algorithms in the ILOG CPLEX Barrier Optimizer for ordering the rows of a matrix:

- automatic, the default, indicated by the value `0`;
- approximate minimum degree (AMD), indicated by the value `1`;
- approximate minimum fill (AMF) indicated by the value `2`;
- nested dissection (ND) indicated by the value `3`.

The log file, as explained in *Ordering-Algorithm Time in the Log File* on page 231, records the time spent by the ordering algorithm in a barrier optimization, so you can experiment with different ordering algorithms and compare their performance on your problem.

Automatic ordering, the default option, will usually be the best choice. This option attempts to choose the most effective of the available ordering methods, and it usually results in the best order. It may require more time than the other settings. The ordering time is usually small relative to the total solution time, and a better order can lead to a smaller total solution time. In other words, a change in this parameter is unlikely to improve performance very much.

The AMD algorithm provides good quality order within moderate ordering time. AMF usually provides better order than AMD (usually 5-10% smaller factors) but it requires somewhat more time (10-20% more). ND often produces significantly better order than AMD or AMF. Ten-fold reductions in runtimes of the ILOG CPLEX Barrier Optimizer have been observed with it on some problems. However, ND sometimes produces worse order, and it requires much more time.

To select an ordering algorithm, set the parameter `BarOrder` to a value 0, 1, 2, or 3.

Using a Starting-Point Heuristic

ILOG CPLEX supports several different heuristics to compute the starting point for the ILOG CPLEX Barrier Optimizer. The starting-point heuristic is determined by the

`BarStartAlg` parameter, and Table 9.8 summarizes the possible settings and their meanings.

Table 9.8 *BarStartAlg Parameter Settings for Starting-Point Heuristics*

Setting	Heuristic
1	dual is 0 (default)
2	estimate dual
3	average primal estimate, dual 0
4	average primal estimate, estimate dual

For most problems the default works well. However, if you are using the dual preprocessing option (setting the parameter `PreDual` to `1`) then one of the other heuristics for computing a starting point may perform better than the default.

- In the Interactive Optimizer, use the command `set barrier startalg i`, substituting a value for `i`.
- When using the Component Libraries, set the parameter `IloCplex::BarStartAlg` or `CPX_PARAM_BARSTARTALG`.

Overcoming Numerical Difficulties

As noted in *Differences between Barrier and Simplex Optimizers* on page 227, the algorithms in the barrier optimizer have very different numerical properties from those in the simplex optimizer. While the barrier optimizer is often extremely fast, particularly on very large problems, numerical difficulties occasionally arise with it in certain classes of problems. For that reason, it is a good idea to run simplex optimizers in conjunction with the barrier optimizer to verify solutions. At its default settings, the ILOG CPLEX Barrier Optimizer always crosses over after a barrier solution to a simplex optimizer, so this verification occurs automatically.

Difficulties in the Quality of Solution

Understanding Solution Quality from the Barrier LP Optimizer on page 233 lists the items that ILOG CPLEX displays about the quality of a barrier solution. If the ILOG CPLEX Barrier Optimizer terminates its work with a solution that does not meet your quality requirements, you can adjust parameters that influence the quality of a solution. Those adjustments affect the choice of barrier algorithm, the limit on barrier corrections, and the choice of starting-point heuristic—topics introduced in *Tuning Barrier Optimizer Performance* on page 235 and recapitulated here in the following subsections.

Change the Barrier Algorithm

The ILOG CPLEX Barrier Optimizer implements the algorithms listed in Table 9.9. The selection of barrier algorithm is controlled by the `BarAlg` parameter. The default option invokes option `3` for LPs and QPs, option `1` for QCPs, and option `1` for MIPs where the ILOG CPLEX Barrier Optimizer is used on the subproblems. Naturally, the default is the fastest for most problems, but it may not work well on LP or QP problems that are primal infeasible or dual infeasible. Options `1` and `2` in the ILOG CPLEX Barrier Optimizer implement a barrier algorithm that also detects infeasibility. (They differ from each other in how they compute a starting point.) Though they are slower than the default option, in a problem demonstrating numerical difficulties, they may eliminate the numerical difficulties and thus improve the quality of the solution.

Table 9.9 *BarAlg Parameter Settings for Barrier Optimizer Algorithm*

BarAlg Setting	Meaning
`0`	default
`1`	algorithm starts with infeasibility estimate
`2`	algorithm starts with infeasibility constant
`3`	standard barrier algorithm

Change the Limit on Barrier Corrections

The default barrier algorithm in the ILOG CPLEX Barrier Optimizer computes an estimate of the maximum number of *centering corrections* that ILOG CPLEX should make on each iteration. You can see this computed value by setting barrier display level two, as explained in *Interpreting the Barrier Log File* on page 230, and checking the value of the parameter to limit corrections. (Its default value is `-1`.) If you see that the current value is `0` (zero), then you should experiment with greater settings. Setting the parameter `BarMaxCor` to a value greater than `0` may improve numerical performance, but there may also be an increase in computation time.

Choose a Different Starting-Point Heuristic

As explained in *Using a Starting-Point Heuristic* on page 238, the default starting-point heuristic works well for most problems suitable to barrier optimization. But for a model that is exhibiting numerical difficulty it is possible that setting the `BarStartAlg` to select a different starting point will make a difference. However, if you are preprocessing your problem as dual (for example, in the Interactive Optimizer you issued the command `set preprocessing dual`), then a different starting-point heuristic may perform better than the default. To change the starting-point heuristic, see Table 9.8 on page 239.

Difficulties during Optimization

Numerical difficulties can degrade performance of the ILOG CPLEX Barrier Optimizer or even prevent convergence toward a solution. There are several possible sources of numerical difficulties:

- elimination of too many dense columns may cause numerical instability;
- tight convergence tolerance may aggravate small numerical inconsistencies in a problem;
- unbounded optimal faces may remain undetected and thus prevent convergence.

The following subsections offer guidance about overcoming those difficulties.

Numerical Instability Due to Elimination of Too Many Dense Columns

Detecting and Eliminating Dense Columns on page 237 explains how to change parameters to encourage ILOG CPLEX to detect and eliminate as many dense columns as possible. However, in some problems, if ILOG CPLEX removes too many dense columns, it may cause numerical instability.

You can check how many dense columns ILOG CPLEX removes by looking at the preprocessing statistics at the beginning of the log file, as explained in *Preprocessing in the Log File* on page 231. If you observe that the removal of too many dense columns results in numerical instability in your problem, then increase the column nonzeros parameter, `BarColNz`.

The default value of the column nonzeros parameter is `0` (zero); that value tells ILOG CPLEX to calculate the parameter automatically.

To see the current value of the column nonzeros parameter—either one you have set or one ILOG CPLEX has automatically calculated—you need to look at the level two display, by setting the `BarDisplay` parameter to `2`.

If you determine that the current value of the column nonzeros parameter is inappropriate for your problem and thus tells ILOG CPLEX to remove too many dense columns, then you can *increase* the parameter `BarColNz` to keep the number of dense columns removed *low.*

Small Numerical Inconsistencies and Tight Convergence Tolerance

If your problem contains small numerical inconsistencies, it may be difficult for the ILOG CPLEX Barrier Optimizer to achieve a satisfactory solution at the default setting of the complementarity convergence tolerance. In such a case, you should increase the convergence tolerance parameter (`BarEpComp` for LP or QP models, `BarQCPEpComp` for QCP models).

Unbounded Variables and Unbounded Optimal Faces

An *unbounded optimal face* occurs in a model that contains a sequence of optimal solutions, all with the same value for the objective function and unbounded variable values. The

ILOG CPLEX Barrier Optimizer will fail to terminate normally if an undetected unbounded optimal face exists.

Normally, the ILOG CPLEX Barrier Optimizer uses its barrier growth parameter, `BarGrowth`, to detect such conditions. If this parameter is increased beyond its default value, the ILOG CPLEX Barrier Optimizer will be less likely to determine that the problem has an unbounded optimal face and more likely to encounter numerical difficulties.

Consequently, you should change the barrier growth parameter *only* if you find that the ILOG CPLEX Barrier Optimizer is terminating its work before it finds the true optimum because it has falsely detected an unbounded face.

Difficulties with Unbounded Problems

ILOG CPLEX detects unbounded problems in either of two ways:

- either it finds a solution with small complementarity that is not feasible for either the primal or the dual formulation of the problem;
- or the iterations tend toward infinity with the objective value becoming very large in absolute value.

The ILOG CPLEX Barrier Optimizer stops when the absolute value of either the primal or dual objective exceeds the objective range parameter, `BarObjRng`.

If you increase the value of `BarObjRng`, then the ILOG CPLEX Barrier Optimizer will iterate more times before it decides that the current problem suffers from an unbounded objective value.

If you know that your problem has *large objective values*, consider increasing `BarObjRng`.

Also if you know that your problem has *large objective values*, consider changing the barrier algorithm by resetting the `BarAlg` parameter.

Diagnosing Barrier Optimizer Infeasibility

When the ILOG CPLEX Barrier Optimizer terminates and reports an infeasible solution, all the usual solution information is available. However, the solution values, reduced costs, and dual variables reported then do not correspond to a basis; hence, that information does not have the same meaning as the corresponding output from the ILOG CPLEX simplex optimizers.

Actually, since the ILOG CPLEX Barrier Optimizer works in a single phase, all reduced costs and dual variables are calculated in terms of the *original* objective function.

If the ILOG CPLEX Barrier Optimizer reports to you that a problem is infeasible, but you still need a basic solution for the problem, use the primal simplex optimizer. ILOG CPLEX

will then use the solution provided by the barrier optimizer to determine a starting basis for the primal simplex optimizer. When the primal simplex optimizer finishes its work, you will have an infeasible basic solution for further infeasibility analysis.

If the default algorithm in the ILOG CPLEX Barrier Optimizer determines that your problem is primal infeasible or dual infeasible, then try the alternate algorithms in the barrier optimizer. These algorithms, though slower than the default, are better at detecting primal and dual infeasibility.

To select one of the barrier infeasibility algorithms, set the `BarAlg` parameter to either `1` or `2`.

Finding a Set of Irreducibly Inconsistent Constraints on page 204 explained how to invoke the infeasibility finder on a solution basis found by one of the simplex optimizers. If you are using the *pure* barrier optimizer (that is, with no crossover to a simplex optimizer), then it will not generate a basis on which you can call the infeasibility finder to analyze your constraints and locate an IIS. Consequently, if you are interested in finding an IIS for your problem, you should invoke the ILOG CPLEX Barrier Optimizer with crossover, as explained in *Controlling Crossover* on page 229.

10

Solving Network-Flow Problems

This chapter tells you more about the ILOG CPLEX Network Optimizer. It includes information about:

- *Choosing an Optimizer: Network Considerations* on page 245
- *Formulating a Network Problem* on page 246
- *Example: Network Optimizer in the Interactive Optimizer* on page 247
- *Example: Using the Network Optimizer with the Callable Library netex1.c* on page 251
- *Solving Network-Flow Problems as LP Problems* on page 258
- *Example: Network to LP Transformation netex2.c* on page 259
- *Solving Problems with the Network Optimizer* on page 249

Choosing an Optimizer: Network Considerations

As explained in *Using the Callable Library in an Application* on page 115, to exploit ILOG CPLEX in your own application, you must first create an ILOG CPLEX environment, instantiate a problem object, and populate the problem object with data. As your next step, you call a ILOG CPLEX optimizer.

If part of your problem is structured as a network, then you may want to consider calling the ILOG CPLEX Network Optimizer. This optimizer may have a positive impact on performance. There are two alternative ways of calling the network optimizer:

- If your problem is an LP where a large part is a network structure, you may call the network optimizer for the populated LP object.
- If your entire problem consists of a network flow, you should consider creating a *network object* instead of an LP object. Then populate it, and solve it with the network optimizer. This alternative generally yields the best performance because it does not incur the overhead of LP data structures. This option is available only for the C library.

How much performance improvement you observe between using only a simplex optimizer versus using the network optimizer followed by either of the simplex optimizers depends on the number and nature of the other constraints in your problem. On a pure network problem, performance has been measured as 100 times faster with the network optimizer. However, if the network component of your problem is small relative to its other parts, then using the solution of the network part of the problem as a starting point for the remainder may or may not improve performance, compared to running the primal or dual simplex optimizer. Only experiments with your own problem can tell.

Formulating a Network Problem

A *network-flow* problem finds the minimal-cost flow through a *network*, where a network consists of a set N of nodes and a set A of arcs connecting the nodes. An arc a in the set A is an ordered pair (i, j) where i and j are nodes in the set N; node i is called the *tail* or the from-node and node j is called the *head* or the to-node of the arc a. Not all the pairs of nodes in a set N are necessarily connected by arcs in the set A. More than one arc may connect a pair of nodes; in other words, $a_1 = (i, j)$ and $a_2 = (i, j)$ may be two different arcs in A, both connecting the nodes i and j in N.

Each arc a may be associated with four values:

- x_a is the flow value, that is, the amount passing through the arc a from its tail (or from-node) to its head (or to-node). The flow values are the modeling variables of a network-flow problem. Negative values are allowed; a negative flow value indicates that there is flow from the head to the tail.
- l_a, the lower bound, determines the minimum flow allowed through the arc a. By default, the lower bound on an arc is 0 (zero).
- u_a, the upper bound, determines the maximum flow allowed through the arc a. By default, the upper bound on an arc is positive infinity.
- c_a, the objective value, determines the contribution to the objective function of one unit of flow through the arc.

Each node n is associated with one value:

- s_n is the supply value at node n.

By convention, a node with strictly positive supply value (that is, $s_n > 0$) is called a *supply* node or a *source*, and a node with strictly negative supply value (that is, $s_n < 0$) is called a *demand* node or a *sink*. A node where $s_n = 0$ is called a *transshipment* node. The sum of all supplies must match the sum of all demands; if not, then the network flow problem is *infeasible*.

T_n is the set of arcs whose tails are node *n*; H_n is the set of arcs whose heads are node *n*. The usual form of a network problem looks like this:

Minimize (or maximize) $$\sum_{a \in A} (c_a x_a)$$

subject to $$\sum_{a \in T_n} x_a - \sum_{a \in H_n} x_a = s_n \forall (n \in N)$$

with these bounds $$l_a \leq x_a \leq u_a \forall (a \in A)$$

That is, for each node, the net flow entering and leaving the node must equal its supply value, and all flow values must be within their bounds. The solution of a network-flow problem is an assignment of flow values to arcs (that is, the modeling variables) to satisfy the problem formulation. A flow that satisfies the constraints and bounds is *feasible*.

Example: Network Optimizer in the Interactive Optimizer

This example is based on a network where the aim is to minimize cost and where the flow through the network has both cost and capacity. Figure 10.1 shows you the nodes and arcs of this network. The nodes are labeled by their identifying node number from 1 through 8. The number inside a node indicates its supply value; 0 (zero) is assumed where no number is given. The arcs are labeled 1 through 14. The lower bound `l`, upper bound `u`, and objective value `c` of each arc are displayed in parentheses `(l, u, c)` beside each arc. In this example, node 1 and node 5 are sources, representing a positive net flow, whereas node 4 and node 8 are sinks, representing negative net flow.

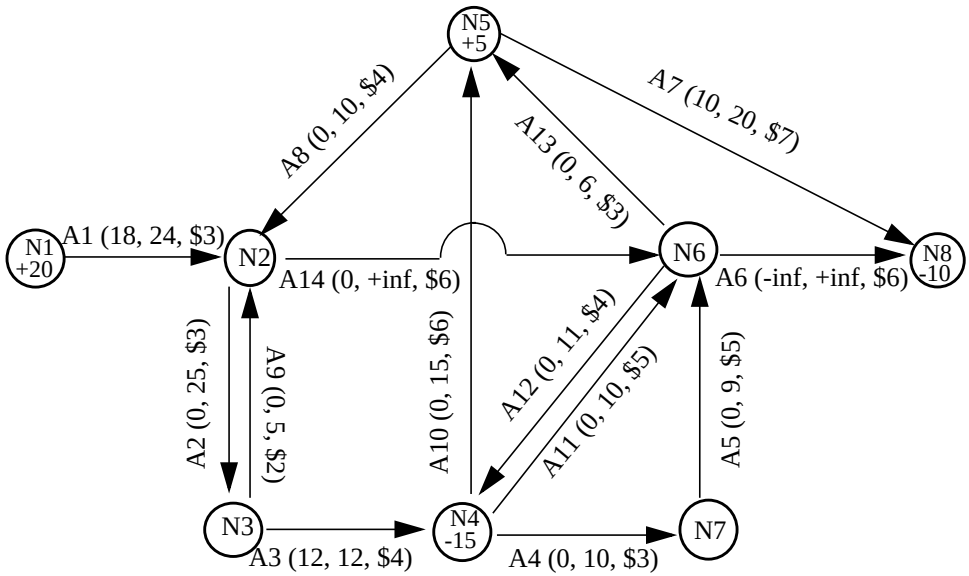

Figure 10.1 *A Directed Network with Arc-Capacity, Flow-Cost, Sinks, and Sources*

In the standard distribution of ILOG CPLEX, the file `nexample.net` contains the formatted problem formulation for this example. You can read it into the Interactive Optimizer with the command `read nexample.net`. After you read the problem into the Interactive Optimizer, you can solve it with the command `netopt` or the command `optimize`.

Understanding the Network Log File

As ILOG CPLEX solves the problem, it produces a log like the following lines:

```
Iteration log . . .
Iteration:     0   Infeasibility     =            48.000000 (5.15396e+13)

Network - Optimal:  Objective =    2.6900000000e+02
Solution time =    0.00 sec.  Iterations = 7 (7)
```

This network log file differs slightly from the log files produced by other ILOG CPLEX optimizers: it contains values enclosed in parentheses that represent modified objective function values.

As long as the network optimizer has not yet found a feasible solution, it is in Phase I. In Phase I, the network optimizer uses modified objective coefficients that penalize infeasibility. At its default settings, the ILOG CPLEX Network Optimizer displays the value of the objective function calculated in terms of these modified objective coefficients in parentheses in the network log file.

You can control the amount of information recorded in the network log file, just as you control the amount of information in other ILOG CPLEX log files. To record no information at all in the log file, use the command `set network display 0`. To display the current objective value in parentheses relative to the actual unmodified objective coefficients, use the command `set network display 1`. To see the display mentioned earlier in this section, leave the network display parameter at its default value, `2`. (If you have changed the default value, you can reset it with the command `set network display 2`.)

Tuning Performance of the Network Optimizer

The default values of parameters controlling the network optimizer are generally the best choices for effective performance. However, the following sections indicate parameters that you may want to experiment with in your particular problem.

Controlling Tolerance

You control the feasibility tolerance for the network optimizer through the parameter `NetEpRHS`. Likewise, you control the optimality tolerance for the network optimizer through the parameter `NetEpOpt`.

Selecting a Pricing Algorithm for the Network Optimizer

On the rare occasions when the network optimizer seems to take too long to find a solution, you may want to change the pricing algorithm to try to speed up computation. The pricing algorithm for the network optimizer is controlled by parameter `NetPPriInd`. All the choices use variations of partial reduced-cost pricing.

Limiting Iterations in the Network Optimizer

Use the parameter `NetItLim` if you want to limit the number of iterations that the network optimizer performs.

Solving Problems with the Network Optimizer

You instruct ILOG CPLEX to apply the network optimizer for solving the LP at hand by setting the `CPX_PARAM_LPMETHOD` parameter to `CPX_ALG_NET` in the Callable Library, or by setting the `RootAlg` parameter to `Network` in Concert Technology. When you do so, ILOG CPLEX performs a sequence of steps. It first searches for a part of the LP that conforms to network structure. Such a part is known as an *embedded* network. It then uses the network optimizer to solve that embedded network. Next, it uses the resulting basis to construct a starting basis for the full LP problem. Finally, it solves the LP problem with a simplex optimizer.

You can also use the network optimizer when solving QPs. To do so using the Callable Library, you set the `CPX_PARAM_QPMETHOD` parameter to `CPX_ALG_NET`. For Concert Technology, the `RootAlg` parameter must be set to `Network`. When ILOG CPLEX uses the

network optimizer to solve a QP, it first ignores the quadratic term and uses the network optimizer to solve the resulting LP. ILOG CPLEX then uses the resulting basis to start a simplex algorithm on the QP model with the original quadratic objective.

Network Extraction

The ILOG CPLEX network extractor searches an LP constraint matrix for a submatrix with the following characteristics:

- the coefficients of the submatrix are all *0* (zero), *1* (one), or *-1 (minus one)*;
- each variable appears in at most two rows with at most one coefficient of *+1* and at most one coefficient of *-1*.

ILOG CPLEX can perform different levels of extraction. The level it performs depends on the `NetFind` parameter.

- When the `NetFind` parameter is set to `1` (one), ILOG CPLEX extracts only the obvious network; it uses no scaling; it scans rows in their natural order; it stops extraction as soon as no more rows can be added to the network found so far.
- When the `NetFind` parameter is set to `2`, the default setting, ILOG CPLEX also uses reflection scaling (that is, it multiplies rows by `-1`) in an attempt to extract a larger network.
- When the `NetFind` parameter is set to `3`, ILOG CPLEX uses general scaling, rescaling both rows and columns, in an attempt to extract a larger network.

In terms of total solution time expended, it may or may not be advantageous to extract the largest possible network. Characteristics of your problem will determine the tradeoff between network size and the number of simplex iterations required to finish solving the model after solving the embedded network.

Even if your problem does not conform precisely to network conventions, the network optimizer may still be advantageous to use. When it is possible to transform the original statement of a linear program into network conventions by these algebraic operations:

- changing the signs of coefficients,
- multiplying constraints by constants,
- rescaling columns,
- adding or eliminating redundant relations,

then ILOG CPLEX will carry out such transformations automatically if you set the `NetFind` parameter appropriately.

Preprocessing and the Network Optimizer

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

Example: Using the Network Optimizer with the Callable Library netex1.c

In the standard distribution of ILOG CPLEX, the file `netex1.c` contains code that creates, solves, and displays the solution of the network-flow problem illustrated in Figure 10.1 on page 248.

Briefly, the `main` function initializes the ILOG CPLEX environment and creates the problem object; it also calls the optimizer to solve the problem and retrieves the solution.

In detail, `main` first calls the Callable Library routine `CPXopenCPLEX`. As explained in *Initialize the ILOG CPLEX Environment* on page 115, `CPXopenCPLEX` must always be the first ILOG CPLEX routine called in a ILOG CPLEX Callable Library application. Those routines create the ILOG CPLEX environment and return a pointer (called `env`) to it. This pointer will be passed to every Callable Library routine. If this initialization routine fails, `env` will be `NULL` and the error code indicating the reason for the failure will be written to `status`. That error code can be transformed into a string by the Callable Library routine `CPXgeterrorstring`.

After `main` initializes the ILOG CPLEX environment, it uses the Callable Library routine `CPXsetintparam` to turn on the ILOG CPLEX screen indicator parameter `CPX_PARAM_SCRIND` so that ILOG CPLEX output appears on screen. If this parameter is turned off, ILOG CPLEX does not produce viewable output, neither on screen, nor in a log file. It is a good idea to turn this parameter on when you are debugging your application.

The Callable Library routine `CPXNETcreateprob` creates an *empty* problem object, that is, a minimum-cost network-flow problem with no arcs and no nodes.

The function `buildNetwork` populates the problem object; that is, it loads the problem data into the problem object. Pointer variables in the example are initialized as `NULL` so that you can check whether they point to valid data—a good programming practice. The most important calls in this function are to the Callable Library routines, `CPXNETaddnodes`, which adds nodes with the specified supply values to the network problem, and `CPXNETaddarcs`, which adds the arcs connecting the nodes with the specified objective values and bounds. In this example, both routines are called with their last argument `NULL` indicating that no names are assigned to the network nodes and arcs. If you want to name arcs and nodes in your problem, pass an array of strings instead.

The function `buildNetwork` also includes a few routines that are not strictly necessary to this example, but illustrate concepts you may find useful in other applications. To delete a

node and all arcs dependent on that node, it uses the Callable Library routine `CPXNETdelnodes`. To change the objective sense to minimization, it uses the Callable Library routine `CPXNETchgobjsen`.

Also `buildNetwork` sets the row growth and column growth parameters `CPX_PARAM_ROWGROWTH` and `CPX_PARAM_COLGROWTH`. These parameters specify the amount that internal arrays are extended if more nodes or arcs are added than currently fit in allocated memory. If you build up a problem by adding nodes and arcs one by one, and if these parameters are set to a low value, then internal arrays will be frequently reallocated; frequent reallocation may negatively impact performance. Ideally, these parameters are set to the maximum number of nodes and arcs that the problem will ever have. This setting will avoid all reallocation and therefore provide best performance. The parameter `CPX_PARAM_ROWGROWTH` pertains to adding nodes to a network problem (and rows to an LP, QP, or MIP problem) whereas `CPX_PARAM_COLGROWTH` pertains to adding arcs to a network problem (or columns to an LP, QP, or MIP problem).

Look again at `main`, where it actually calls the network optimizer with the Callable Library routine, `CPXNETprimopt`. If `CPXNETprimopt` returns a nonzero value, then an error has occurred; otherwise, the optimization was successful. Before retrieving that solution, it is necessary to allocate arrays to hold it. Then use `CPXNETsolution` to copy the solution into those arrays. After displaying the solution on screen, write the network problem into a file, `netex1.net` in the NET file format.

The `TERMINATE`: label is used as a place for the program to exit if any type of error occurs. Therefore, code following this label cleans up: it frees the memory that has been allocated for the solution data; it frees the network object by calling `CPXNETfreeprob`; and it frees the ILOG CPLEX environment by calling `CPXcloseCPLEX`. All freeing should be done only if the data is actually available. The Callable Library routine `CPXcloseCPLEX` should always be the last ILOG CPLEX routine called in a ILOG CPLEX Callable Library application. In other words, all ILOG CPLEX objects that have been allocated should be freed before the call to `CPXcloseCPLEX`.

The complete program, `netex1.c`, appears here and online in the standard distribution.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/netex1.c                                             */
/*  Version 9.0                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2003 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* netex1.c - Entering and optimizing a network problem */

/* Import the CPLEX function declarations and the C library
   header file stdio.h with the include of cplex.h. */

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

```
#include <stdlib.h>

/* Import the declarations for the string functions */

#include <string.h>



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

static int
   buildNetwork  (CPXENVptr env, CPXNETptr net);

static void
   free_and_null (char **ptr);


int
main (void)
{
   /* Declare variables and arrays for retrieving problem data and
      solution information later on. */

   int      narcs;
   int      nnodes;
   int      solstat;
   double   objval;
   double   *x     = NULL;
   double   *pi    = NULL;
   double   *slack = NULL;
   double   *dj    = NULL;

   CPXENVptr env = NULL;
   CPXNETptr net = NULL;
   int       status;
   int       i, j;

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

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

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

   /* Turn on output to the screen */
```

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

/* Create the problem. */

net = CPXNETcreateprob (env, &status, "netex1");

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

if ( net == NULL ) {
   fprintf (stderr, "Failed to create network object.\n");
   goto TERMINATE;
}

/* Fill in the data for the problem.  Note that since the space for
   the data already exists in local variables, we pass the arrays
   directly to the routine to fill in the data structures.  */

status = buildNetwork (env, net);

if ( status ) {
   fprintf (stderr, "Failed to build network problem.\n");
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */

status = CPXNETprimopt (env, net);
if ( status ) {
   fprintf (stderr, "Failed to optimize network.\n");
   goto TERMINATE;
}

/* get network dimensions */

narcs  = CPXNETgetnumarcs  (env, net);
nnodes = CPXNETgetnumnodes (env, net);

/* allocate memory for solution data */

x     = (double *) malloc (narcs  * sizeof (double));
dj    = (double *) malloc (narcs  * sizeof (double));
pi    = (double *) malloc (nnodes * sizeof (double));
slack = (double *) malloc (nnodes * sizeof (double));

if ( x     == NULL ||
     dj    == NULL ||
     pi    == NULL ||
     slack == NULL   ) {
```

```
      fprintf (stderr, "Failed to allocate arrays.\n");
      goto TERMINATE;
   }

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

   /* Write the output to the screen. */

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

   for (i = 0; i < nnodes; i++) {
      printf ("Node %2d:  Slack = %10f  Pi = %10f\n", i, slack[i], pi[i]);
   }

   for (j = 0; j < narcs; j++) {
      printf ("Arc  %2d:  Value = %10f  Reduced cost = %10f\n",
              j, x[j], dj[j]);
   }

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

   status = CPXNETwriteprob (env, net, "netex1.net", NULL);
   if ( status ) {
      fprintf (stderr, "Failed to write network to disk.\n");
      goto TERMINATE;
   }

TERMINATE:

   /* Free memory for solution data */

   free_and_null ((char **) &x);
   free_and_null ((char **) &dj);
   free_and_null ((char **) &pi);
   free_and_null ((char **) &slack);

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

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

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

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

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
```

```
            be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

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

      return (status);

}  /* END main */



static int
buildNetwork (CPXENVptr env, CPXNETptr net)
{
   int status = 0;

   /* definitions to improve readability */

#  define NNODES  8
#  define NARCS  14
#  define inf    CPX_INFBOUND

   /* Define list of supply values for the nodes */

   double supply[NNODES] = {20.0, 0.0, 0.0, -15.0, 5.0, 0.0, 0.0, -10.0};

   /* Define list of tail or from-node indices as well as head or
      to-node indices for the arcs.  Notice that according to C
      standard the first node has index 0. */

   int    tail[NARCS] = {   0,    1,    2,    3,    6,    5,    4,
                            4,    2,    3,    3,    5,    5,    1};
   int    head[NARCS] = {   1,    2,    3,    6,    5,    7,    7,
                            1,    1,    4,    5,    3,    4,    5};

   /* Define list of objective values and lower and upper bound values
      for the arcs */

   double obj [NARCS] = { 3.0,  3.0,  4.0,  3.0,  5.0,  6.0,  7.0,
                          4.0,  2.0,  6.0,  5.0,  4.0,  3.0,  6.0};
   double ub  [NARCS] = {24.0, 25.0, 12.0, 10.0,  9.0,  inf, 20.0,
                         10.0,  5.0, 15.0, 10.0, 11.0,  6.0,  inf};
   double lb  [NARCS] = {18.0,  0.0, 12.0,  0.0,  0.0, -inf,  0.0,
                          0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0};

   /* Delete existing network.  This is not necessary in this
      context since we know we have an empty network object.
      Notice that CPXNETdelnodes deletes all arcs incident to
      the deleted nodes as well.  Therefore this one function
      call effectively deletes an existing network problem. */

   if ( CPXNETgetnumnodes (env, net) > 0 ) {
      status = CPXNETdelnodes (env, net, 0,
```

```
                                    CPXNETgetnumnodes (env, net)-1);
      if ( status ) goto TERMINATE;
   }

   /* Set growth rates for rows/nodes and columns/arcs.  This
      is to avoid internal memory reallocations while adding
      nodes and arcs.  Since we are adding all nodes and all
      arcs using only one function call for each it is actually
      unnecessary, but if more function calls are used, finding
      the right settings may improve performance. */

   status = CPXsetintparam (env, CPX_PARAM_ROWGROWTH, NNODES);
   if ( status ) goto TERMINATE;

   status = CPXsetintparam (env, CPX_PARAM_COLGROWTH, NARCS);
   if ( status ) goto TERMINATE;

   /* Set optimization sense */

   status = CPXNETchgobjsen (env, net, CPX_MIN);
   if ( status ) goto TERMINATE;

   /* Add nodes to network along with their supply values,
      but without any names. */

   status = CPXNETaddnodes (env, net, NNODES, supply, NULL);
   if ( status ) goto TERMINATE;

   /* Add arcs to network along with their objective values and
      bounds, but without any names. */

   status = CPXNETaddarcs (env, net, NARCS, tail, head, lb, ub, obj, NULL);
   if ( status ) goto TERMINATE;

TERMINATE:

   return (status);

}  /* END buildnetwork */


static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */
```

Solving Network-Flow Problems as LP Problems

A network-flow model is an LP model with special structure. The ILOG CPLEX Network Optimizer is a highly efficient implementation of the primal simplex technique adapted to take advantage of this special structure. In particular, no basis factoring occurs. However, it is possible to solve network models using any of the ILOG CPLEX LP optimizers if first, you convert the network data structures to those of an LP model. To convert the network data structures to LP data structures, in the Interactive Optimizer, use the command `change problem lp`; from the Callable Library, use the routine `CPXcopynettolp`.

The LP formulation of our example from Figure 10.1 on page 248 looks like this:

Minimize

$$3a_1 + 3a_2 + 4a_3 + 3a_4 + 5a_5 + 6a_6 + 7a_7 + 4a_8 + 2a_9 + 6a_{10} + 5a_{11} + 4a_{12} + 3a_{13} + 6a_{14}$$

subject to

$$\begin{aligned}
a_1 &= 20\\
-a_1 + a_2 - a_8 - a_9 + a_{14} &= 0\\
- a_2 + a_3 + a_9 &= 0\\
- a_3 + a_4 + a_{10} + a11 - a_{12} &= -15\\
a_7 + a_8 - a_{10} - a_{13} &= 5\\
- a_5 + a_6 - a_{11} + a_{12} + a_{13} - a_{14} &= 0\\
- a_4 + a_5 &= 0\\
- a_6 - a_7 &= -10
\end{aligned}$$

with these bounds

$$\begin{aligned}
18 \le a_1 &\le 24\\
0 \le a_2 &\le 25\\
a_3 &= 12\\
0 \le a_4 &\le 10\\
0 \le a_5 &\le 9\\
a_6 &\text{ free}\\
0 \le a_7 &\le 20
\end{aligned}$$

$$0 \leq a_8 \leq 10$$

$$0 \leq a_9 \leq 5$$

$$0 \leq a_{10} \leq 15$$

$$0 \leq a_{11} \leq 10$$

$$0 \leq a_{12} \leq 11$$

$$0 \leq a_{13} \leq 6$$

$$0 \leq a_{14}$$

In that formulation, in each column there is exactly one coefficient equal to *1* (one), exactly one coefficient equal to *-1*, and all other coefficients are *0* (zero).

Since a network-flow problem corresponds in this way to an LP problem, you can indeed solve a network-flow problem by means of a ILOG CPLEX LP optimizer as well. If you read a network-flow problem into the Interactive Optimizer, you can transform it into its LP formulation with the command `change problem lp`. After this change, you can apply any of the LP optimizers to this problem.

When you change a network-flow problem into an LP problem, the basis information that is available in the network-flow problem is passed along to the LP formulation. In fact, if you have already solved the network-flow problem to optimality, then if you call the primal or dual simplex optimizers (for example, with the Interactive Optimizer command `primopt` or `tranopt`), that simplex optimizer will perform no iterations.

Generally, you can also use the same basis from a basis file for both the LP and the network optimizers. However, there is one exception: in order to use an LP basis with the network optimizer, at least one slack variable or one artificial variable needs to be basic. *Starting from an Advanced Basis* on page 190 explains more about this topic in the context of LP optimizers.

If you have already read the LP formulation of a problem into the Interactive Optimizer, you can transform it into a network with the command `change problem network`. Given any LP problem and this command, ILOG CPLEX will try to find the largest network embedded in the LP problem and transform it into a network-flow problem. However, as it does so, it discards all rows and columns that are not part of the embedded network. At the same time, ILOG CPLEX passes along as much basis information as possible to the network optimizer.

Example: Network to LP Transformation netex2.c

This example shows how to transform a network-flow problem into its corresponding LP formulation. That example also indicates why you might want to make such a change. The

example reads a network-flow problem from a file (rather than populating the problem object by adding rows and columns as in `netex1.c`). It then attempts to solve the problem by calling the Callable Library routine `CPXNETprimopt`. If it determines that the problem is infeasible, it then transforms the problem into its LP formulation so that the infeasibility finder can analyze the problem and possibly indicate the cause of the infeasibility in an irreducibly inconsistent set (IIS). To perform this analysis, the application calls the Callable Library routine `CPXiiswrite` to write the IIS to the file `netex2.iis`.

The complete program, `netex2.c`, appears here and online in the standard distribution.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/netex2.c                                             */
/*  Version 9.0                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2003 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* netex2.c - Reading and optimizing a network problem.
              Transforming to LP to find IIS. */

/* Import the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

/* Import the declarations for the string functions */

#include <string.h>



int
main (int argc, char **argv)
{
   /* Declare variables and arrays for retrieving problem data and
      solution information later on. */

   int       status = 0;
   CPXENVptr env = NULL;
   CPXNETptr net = NULL;
   CPXLPptr  lp  = NULL;

   /* Check command line */

   if ( argc != 2 ) {
      fprintf (stderr, "Usage: %s <network file>\n", argv[0]);
      fprintf (stderr, "Exiting ...\n");
      goto TERMINATE;
   }

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);
```

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

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

/* Turn on output to the screen */

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

/* Create the problem. */

net = CPXNETcreateprob (env, &status, "netex2");

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

if ( net == NULL ) {
   fprintf (stderr, "Failed to create network object.\n");
   goto TERMINATE;
}

/* Read network problem data from file
   with filename given as command line argument. */

status = CPXNETreadcopyprob (env, net, argv[1]);

if ( status ) {
   fprintf (stderr, "Failed to build network problem.\n");
   goto TERMINATE;
}

/* Optimize the problem */

status = CPXNETprimopt (env, net);
if ( status ) {
   fprintf (stderr, "Failed to optimize network.\n");
   goto TERMINATE;
}
```

```
   /* Check network solution status */

   if ( CPXNETgetstat (env, net) == CPX_STAT_INFEASIBLE ) {

      /* Create LP object used for invoking infeasibility finder */

      lp = CPXcreateprob (env, &status, "netex2");
      if ( lp == NULL ) {
         fprintf (stderr, "Failed to create LP object.\n");
         goto TERMINATE;
      }

      /* Copy LP representation of network problem to lp object, along
         with the current basis available in the network object. */

      status = CPXcopynettolp (env, lp, net);
      if ( status ) {
         fprintf (stderr, "Failed to copy network as LP.\n");
         goto TERMINATE;
      }

      /* Optimize the LP with primal to create an LP solution.  This
         optimization will start from the basis previously generated by
         CPXNETprimopt() as long as the advance indicator is switched
         on (its default).  */

      status = CPXsetintparam (env, CPX_PARAM_LPMETHOD, CPX_ALG_PRIMAL);
      if ( status ) {
         fprintf (stderr,
                  "Failure to set LP method, error %d.\n", status);
         goto TERMINATE;
      }

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

      /* Find IIS and write it to a file */

      status = CPXiiswrite (env, lp, "netex2.iis");
      if ( status ) {
         fprintf (stderr, "Failed to find IIS or write IIS file\n");
         goto TERMINATE;
      }

      printf ("IIS written to file netex2.iis\n");
   }
   else {
      printf ("Network problem not proved to be infeasible\n");
   }

TERMINATE:

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

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

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

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

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

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

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

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

   return (status);

}  /* END main */
```

C H A P T E R

11

Solving Problems with a Quadratic Objective (QP)

This chapter tells you about solving *convex quadratic programming* problems (QPs) with ILOG CPLEX. This chapter contains sections about:

- *Identifying Convex QPs* on page 266
- *Entering QPs* on page 267
- *Saving QP Problems* on page 268
- *Changing Problem Type in QPs* on page 269
- *Changing Quadratic Terms* on page 270
- *Optimizing QPs* on page 271
- *Example: Creating a QP, Optimizing, Finding a Solution* on page 273
- *Example: Reading a QP from a File qpex2.c* on page 286

Identifying Convex QPs

Conventionally, a quadratic program (QP) is formulated this way:

Minimize $\frac{1}{2}\boldsymbol{x}^T\boldsymbol{Q}x + \boldsymbol{c}^T\boldsymbol{x}$

subject to $\boldsymbol{Ax} \sim b$

with these bounds $l \leq \boldsymbol{x} \leq u$

where the relation ~ may be any combination of equal to, less than or equal to, greater than or equal to, or range constraints. As in other problem formulations, l indicates lower and u upper bounds. $\boldsymbol{Q}$ is a matrix of objective function coefficients. That is, the elements Q_{jj} are the coefficients of the quadratic terms x_j^2, and the elements Q_{ij} and Q_{ji} are summed together to be the coefficient of the term $x_i x_j$.

ILOG CPLEX distinguishes two kinds of $\boldsymbol{Q}$ matrices:

- In a *separable* problem, only the diagonal terms of the matrix are defined.
- In a *nonseparable* problem, at least one off-diagonal term of the matrix is nonzero.

ILOG CPLEX can solve *minimization* problems having a *convex* quadratic objective function. Equivalently, it can solve *maximization* problems having a *concave* quadratic objective function. All linear objective functions satisfy this property for both minimization and maximization. However, you cannot always assume this property in the case of a quadratic objective function. Intuitively, recall that that any point on the line between two arbitrary points of a convex function will be above that function. In more formal terms, a continuous segment (that is, a straight line) connecting two arbitrary points on the graph of the objective function will not go below the objective in a minimization, and equivalently, the straight line will not go above the objective in a maximization. Figure 11.1 illustrates this intuitive idea for an objective function in one variable. It is possible for a quadratic function in more than one variable to be neither convex nor concave.

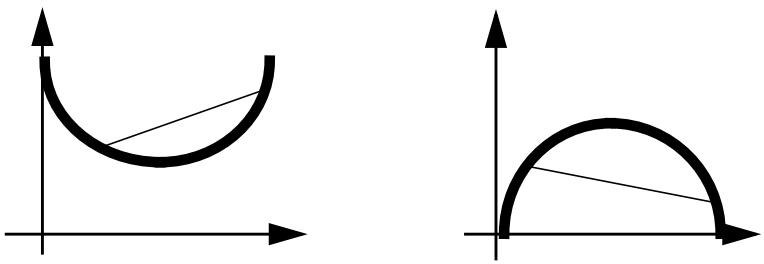

Figure 11.1 *Minimize a Convex Objective Function, Maximize a Concave Objective Function*

The question of whether a quadratic objective function is convex or concave is equivalent to whether the matrix $\boldsymbol{Q}$ is positive semi-definite or negative semi-definite. For convex QPs, $\boldsymbol{Q}$

must be *positive semi-definite*; that is, $\boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x} \geq 0$ for every vector $\boldsymbol{x}$, whether or not $\boldsymbol{x}$ is feasible. For concave maximization problems, the requirement is that $\boldsymbol{Q}$ must be *negative semi-definite*; that is, $\boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x} \leq 0$ for every vector $\boldsymbol{x}$. It is conventional to use the same term, positive semi-definite, abbreviated PSD, for both cases, on the assumption that a maximization problem with a negative semi-definite $\boldsymbol{Q}$ can be transformed into an equivalent PSD.

For a separable function, it is sufficient to check whether the individual diagonal elements of the matrix $\boldsymbol{Q}$ are of the correct sign. For a nonseparable case, it may be less easy to determine in advance the convexity of $\boldsymbol{Q}$. However, ILOG CPLEX determines this property during the early stages of optimization and terminates if the quadratic objective term in a QP is found to be not PSD.

For a more complete explanation of quadratic programming generally, a text, such as one of those listed in *Further Reading* on page 36 of the preface of this manual, will be helpful.

Entering QPs

ILOG CPLEX supports two views of quadratic objective functions, a matrix view and an algebraic view.

In the matrix view, commonly found in textbook presentations of QP, the objective function is defined as $1/2\ \boldsymbol{x}^T\boldsymbol{Q}x + \boldsymbol{c}^T\boldsymbol{x}$, where $\boldsymbol{Q}$ must be symmetric and positive semi-definite for a minimization problem, or negative semi-definite for a maximization problem. This view is supported by the MPS file format and the Callable Library routines, where the quadratic objective function information is specified by providing the matrix $\boldsymbol{Q}$. Thus, by definition, the factor of 1/2 must be considered when entering a model using the matrix view, as it will be implicitly assumed by the optimization routines. Similarly, symmetry of the $\boldsymbol{Q}$ matrix data is required; the MPS reader will return an error status code if the file contains unequal off-diagonal components, such as a nonzero value for one and zero (or omitted) for the other.

In the algebraic view, a quadratic objective function is specified as an expressions of the form:

```
c1*x1 + ... + cn*xn + q11*x1*x1 + q12*x1*x2 + ... + qnn*xn*xn.
```

This view is supported by the LP format, when entering quadratic objective functions in the Interactive Optimizer, and by Concert Technology. Again, a quadratic objective function must be convex in the case of a minimization problem, or concave in the case of a maximization problem. When entering a quadratic objective with the algebraic view, neither symmetry considerations nor any implicit factors need to be considered, and indeed attempting to specify both of the off-diagonal elements for one of the quadratic terms may result in double the intended value of the coefficient.

Examples:

ILOG CPLEX LP format requires the factor of 1/2 to be explicitly specified in the file.

```
Minimize
 obj: [ 100 x1 ^2 - 200 x1 * x2 + 100 x2 ^2 ] / 2
```

MPS format for this same objective function would contain the following.

```
QMATRIX
    x1        x1                100
    x1        x2               -100
    x2        x1               -100
    x2        x2                100
```

A C++ Concert program having such an objective function might include the following.

```
model.add(IloMinimize(env, 0.5 * (100*x[0]*x[0] + 100*x[1]*x[1]
                                   - 200*x[0]*x[1] ) ));
```

Or since the algebraic view is supported, the factor of could be simplified as in the following equivalent expression:

```
model.add(IloMinimize(env,   (50*x[0]*x[0] + 50*x[1]*x[1]
                                   - 100*x[0]*x[1] ) ));
```

A similar Java program using Concert might express it this way:

```
IloNumExpr x00 = model.prod(100, x[0], x[0]);
IloNumExpr x11 = model.prod(100, x[1], x[1]);
IloNumExpr x01 = model.prod(-200, x[0], x[1]);
IloNumExpr Q   = model.prod(0.5, model.sum(x00, x11, x01));
model.add(model.minimize(Q));
```

Again, the user could choose to simplify the above expression algebraically if that suits the purposes of the application better.

Finally, a Callable Library application in C might construct the quadratic objective function in a way similar to the following:

```
zqmatind[0] = 0;     zqmatind[2] = 0;
zqmatval[0] = 100.0; zqmatval[2] = -100.0;
zqmatind[1] = 1;     zqmatind[3] = 1;
zqmatval[1] =-100.0; zqmatval[3] = 100.0;
```

To re-emphasize the point about the factor of 1/2 in any of these methods, if that objective function is evaluated with a solution of `x1 = 1.000000` and `x2 = 3.000000`, the result to be expected is `200`, not `400`.

Saving QP Problems

After you enter a QP problem, whether interactively or by reading a formatted file, you can then save the problem in a formatted file. The formats available to you are LP, MPS, and

SAV. When you save a QP problem in one of these formats, the quadratic information will also be recorded in the formatted file.

In addition, you can save the quadratic part of a problem in a QP file (a formatted file with the extension `.qp`, as explained in the reference manual of *ILOG CPLEX File Formats*). To do so:

- In the Interactive Optimizer, use the `write` command
- In the Callable Library, use the routine `CPXqpwrite`.
- Writing a QP format file is not supported by `IloCplex`.

Changing Problem Type in QPs

Concert Technology (that is, applications written in C++ , Java, or .NET with the class `IloCplex`) treats all models as capable of containing quadratic coefficients in the objective function. These coefficients can therefore be added or deleted at will. When extracting a model with a quadratic objective function, `IloCplex` will automatically detect it as a QP and make the required adjustments to data structures.

However, the other ways of using ILOG CPLEX (the Callable Library and the Interactive Optimizer) require an explicit *problem type* to distinguish Linear Programs (LPs) from QPs. The following sections discuss the topic for these users.

When you enter a problem, ILOG CPLEX determines the problem type from the available information. When read from a file (LP, MPS, or SAV format, for example), or entered interactively, a continuous optimization problem is treated as being of type `qp` if quadratic coefficients are present in the objective function. Otherwise, the problem type is `lp`. The issue of problem types that support integer restrictions in conjunction with quadratic variables is discussed in Chapter 13, *Solving Mixed Integer Programming Problems (MIP)*.

If you enter a problem that lacks any quadratic coefficients, its Problem Type is initially `lp`. If you then wish to modify the problem to contain quadratic coefficients, you do this by first changing the Problem Type to `qp`. Conversely, if you have entered a QP model and wish to remove all the quadratic coefficients and thus convert the model to an LP, you can change the Problem Type to `lp`. Note that deleting each of the quadratic coefficients individually still leaves the Problem Type as `qp`, although in most instances the distinction between this problem and its `lp` or `qp` counterpart is somewhat arbitrary in terms of the steps to solve it.

When using the Interactive Optimizer, you use the command `change problem` with one of the following options:

- `lp` indicates that you want ILOG CPLEX to treat the problem as an LP. This change in Problem Type removes from your problem all the quadratic information, if there is any present.

- `qp` indicates that you want ILOG CPLEX to treat the problem as a QP. This change in Problem Type creates in your problem an empty quadratic matrix, if there is not one already present, for the objective function, ready for populating via the `change qpterm` command.

From the Callable Library, use the routine `CPXchgprobtype` to change the Problem Type to either `CPXPROB_LP` or `CPXPROB_QP` for the LP and QP case, respectively, for the same purposes.

Changing Quadratic Terms

ILOG CPLEX distinguishes between a *quadratic algebraic term* and a *quadratic matrix coefficient*. The quadratic algebraic terms are the coefficients that appear in the algebraic expression defined as part of the ILOG CPLEX LP format. The quadratic matrix coefficients appear in $\boldsymbol{Q}$. The quadratic coefficient of an off-diagonal term must be distributed within the $\boldsymbol{Q}$ matrix, and it is always one-half the value of the quadratic algebraic term.

To clarify that terminology, consider this example:

Minimize $a + b + 1/2(a^2 + 4ab + 7b^2)$

subject to $a + b \geq 10$

with these bounds $a \geq 0$ and $b \geq 0$

The off-diagonal quadratic algebraic term in that example is 4, so the quadratic matrix $\boldsymbol{Q}$ is

$$\begin{bmatrix} 1 & 2 \\ 2 & 7 \end{bmatrix}$$

- In a QP, you can change the quadratic matrix coefficients in the Interactive Optimizer by using the command `change qpterm`.
- From the Callable Library, use the routine `CPXchgqpcoef` to change quadratic matrix coefficients.
- Concert Technology does not support direct editing of expressions other than linear expressions. Consequently, to change a quadratic objective function, you need to create an expression with the modified quadratic objective and use the `setExpr` method of `IloObjective` to install it.

Changing an off-diagonal element changes the corresponding symmetric element as well. In other words, if a call to `CPXchgqpcoef` changes $\boldsymbol{Q_{ij}}$ to a value, it also changes $\boldsymbol{Q_{ji}}$ to that same value.

To change the off-diagonal quadratic term from 4 to 6, use this sequence of commands in the Interactive Optimizer:

```
CPLEX> change qpterm
Change which quadratic term ['variable' 'variable']: a b
Present quadratic term of variable 'a', variable 'b' is 4.000000.
Change quadratic term of variable 'a', variable 'b' to what: 6.0
Quadratic term of variable 'a', variable 'b' changed to 6.000000.
```

From the Callable Library, the `CPXchgqpcoef` call to change the off-diagonal term from `4` to `6` would change both of the off-diagonal matrix coefficients from `2` to `3`. Thus, the indices would be `0` and `1`, and the new matrix coefficient value would be 3.

If you have entered a linear problem without any quadratic terms, and you want to create quadratic terms, you must first *change the problem type* to QP. To do so, use the command `change problem qp`. This command will create an empty quadratic matrix with $\boldsymbol{Q} = \mathbf{0}$.

When you change quadratic terms, there are still restrictions on the properties of the $\boldsymbol{Q}$ matrix. In a minimization problem, it must be convex, positive semi-definite. In a maximization problem, it must be concave, negative semi-definite. For example, if you change the sense of an objective function in a convex $\boldsymbol{Q}$ matrix from minimization to maximization, you will thus make the problem unsolvable. Likewise, in a convex $\boldsymbol{Q}$ matrix, if you make a diagonal term negative, you will thus make the problem unsolvable.

Optimizing QPs

ILOG CPLEX allows you to solve your QP models through a simple interface, by calling the default optimizer as follows:

- In the Interactive Optimizer, use the command `optimize`.
- From the Callable Library, use the routine `CPXqpopt`.
- In Concert applications, use the `solve` method of `IloCplex`.

With default settings, this will result in the barrier optimizer being called to solve a continuous QP.

For users who wish to tune the performance of their applications, there are two Simplex optimizers to try for solving QPs. They are Dual Simplex and Primal Simplex. You can also use the Network Simplex optimizer; this solves the model as an LP network (temporarily ignoring the quadratic term in the objective function) and takes this solution as a starting point for the Primal Simplex QP optimizer. This choice of QP optimizer is controlled by the

RootAlg parameter (QPMETHOD in the Interactive Optimizer and in the Callable Library). Table 11.1 shows you the possible settings.

Table 11.1 *RootAlg Parameter Settings*

RootAlg Value	Optimizer
0	Automatic (default)
1	Primal Simplex
2	Dual Simplex
3	Network Simplex
4	Barrier
5	Sifting
6	Concurrent

Many of the optimizer tuning decisions for LP apply in the QP case; and parameters that control Barrier and Simplex optimizers in the LP case can be set for the QP case, although in some instances to differing effect. Most models are solved fastest by default parameter settings. See the LP chapter for tuning advice.

Just as for the LP case, each of the available QP optimizers automatically preprocesses your model, conducting presolution problem analysis and reductions appropriate for a QP.

The Barrier optimizer for QP supports crossover, but unlike other LP optimizers, its crossover step is off by default for QPs. The QP Simplex optimizers return basic solutions, and these bases can be used for purposes of restarting sequences of optimizations, for example. As a result, application writers who wish to allow end users control over the choice of QP optimizer need to be aware of this fundamental difference and to program carefully. For most purposes, the nonbasic Barrier solution is entirely satisfactory, in that all such solutions fully satisfy the standard optimality and feasibility conditions of optimization theory.

Diagnosing QP Infeasibility

Diagnosis of an infeasible QP problem is best done by relaxing it to its associated LP problem, that is, where the ***Q*** matrix has been set to **0**.

To change the Problem Type:

- in the Interactive Optimizer, use the command `change problem lp;`
- from the Callable Library, use the routine `CPXchgprobtype`.

Since `IloCplex` handles Problem Types transparently, the way to diagnose an infeasible model is slightly different. Since infeasibility does not depend on the objective function, you start by removing the objective extractable from the extracted model. This way, the model seen by the `cplex` object is an LP with a 0 objective and the LP IIS finder can be applied. To get the original model back, simply add the objective back to the model.

You can then solve the relaxed linear version by means of the ILOG CPLEX simplex optimizer. Then you can apply the ILOG CPLEX infeasibility finder to that relaxed solution, with its associated, original QP information, to help you diagnose the source of the infeasibility. (*Diagnosing LP Infeasibility* on page 200 explains how to use the ILOG CPLEX infeasibility finder following a simplex optimizer.)

Note that it is possible for the outcome of that analysis to be a confirmation that your model (viewed as an LP) is feasible after all. This is typically a symptom that your QP model is numerically unstable, or ill-conditioned. Unlike the simplex optimizers for LP, the QP optimizers are primal-dual in nature, and one result of that is the scaling of the objective function interacts directly with the scaling of the constraints.

Just as our recommendation regarding numerical difficulties on LP models (see *Numerical Difficulties* on page 196) is for coefficients in the constraint matrix not to vary by more than about six orders of magnitude, for QP this recommendation expands to include the quadratic elements of the objective function coefficients as well. Fortunately, in most instances it is straightforward to scale your objective function, by multiplying or dividing all the coefficients (linear **and** quadratic) by a constant factor, which changes the unit of measurement for the objective but does not alter the meaning of the variables or the sense of the problem as a whole. If your objective function itself contains a wide variation of coefficient magnitudes, you may also want to consider scaling the individual columns to achieve a closer range.

Example: Creating a QP, Optimizing, Finding a Solution

This example shows you how to build and solve a QP. The problem being created and solved is:

Maximize

$$x_1 + 2x_2 + 3x_3 - 0.5\,(33x_1^2 + 22x_2^2 + 11x_3^2 - 12x_1x_2 - 23x_2x_3)$$

subject to

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

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

with these bounds

$$0 \leq x_1 \leq 40$$

$$0 \leq x_2 \leq +\infty$$
$$0 \leq x_3 \leq +\infty$$

Example: iloqpex1.cpp

This example is almost identical to `ilolpex1.cpp` with only function `populatebyrow` to create the model. Also, this function differs only in the creation of the objective from its `ilolpex1.cpp` counterpart. Here the objective function is created and added to the model like this:

```
model.add(IloMaximize(env, x[0] + 2 * x[1] + 3 * x[2]
             - 0.5 * (33*x[0]*x[0] + 22*x[1]*x[1] + 11*x[2]*x[2]
                           - 12*x[0]*x[1] - 23*x[1]*x[2]) ));
```

In general, any expression built of basic operations +, -, *, / constant, and brackets [] that amounts to a quadratic and optional linear term can be used for building QP objective function. Note that, if the expressions of the objective or any constraint of the model contains `IloPiecewiseLinear`, then when a quadratic objective is specified the model becomes an MIQP problem. (Piecewise-linearity is not the only characteristic that renders a model MIQP. See also, for example, the features in *Logical Constraints in Optimization* on page 411, where automatic transformation with logical constraints can render a problem MIQP.)

Complete Program: iloqpex1.cpp

The complete program, `iloqpex1.cpp`, appears here and online in the standard distribution.

```
// ------------------------------------------------------------- -*- C++ -*-
// File: examples/src/iloqpex1.cpp
// Version 9.0
// ------------------------------------------------------------------------
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ------------------------------------------------------------------------
//
// iloqpex1.cpp - Entering and optimizing a quadratic problem.

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

static void
   populatebyrow     (IloModel model, IloNumVarArray var, IloRangeArray con);

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

```
      IloNumVarArray var(env);
      IloRangeArray con(env);

      populatebyrow (model, var, con);

      IloCplex cplex(model);

      // Optimize the problem and obtain solution.
      if ( !cplex.solve() ) {
         env.error() << "Failed to optimize LP" << endl;
         throw(-1);
      }

      IloNumArray vals(env);
      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;
      cplex.getValues(vals, var);
      env.out() << "Values        = " << vals << endl;
      cplex.getSlacks(vals, con);
      env.out() << "Slacks        = " << vals << endl;
      cplex.getDuals(vals, con);
      env.out() << "Duals         = " << vals << endl;
      cplex.getReducedCosts(vals, var);
      env.out() << "Reduced Costs = " << vals << endl;

      cplex.exportModel("qpex1.lp");
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();

   return 0;
}  // END main


// To populate by row, we first create the variables, and then use them to
// create the range constraints and objective.  The model we create is:
//
//    Maximize
//     obj: x1 + 2 x2 + 3 x3
//            - 0.5 ( 33*x1*x1 + 22*x2*x2 + 11*x3*x3
//                              - 12*x1*x2 - 23*x2*x3 )
//    Subject To
//     c1: - x1 + x2 + x3 <= 20
//     c2: x1 - 3 x2 + x3 <= 30
//    Bounds
//     0 <= x1 <= 40
//    End

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

```
   x.add(IloNumVar(env, 0.0, 40.0));
   x.add(IloNumVar(env));
   x.add(IloNumVar(env));
   model.add(IloMaximize(env, x[0] + 2 * x[1] + 3 * x[2]
                           - 0.5 * (33*x[0]*x[0] + 22*x[1]*x[1] +
                                    11*x[2]*x[2] - 12*x[0]*x[1] -
                                    23*x[1]*x[2]                ) ));

   c.add( - x[0] +     x[1] + x[2] <= 20);
   c.add(   x[0] - 3 * x[1] + x[2] <= 30);
   model.add(c);
}  // END populatebyrow
```

Example: QPex1.java

This example is almost identical to `LPex1.java` using only the function `populatebyrow` to create the model. Also, this function differs only in the creation of the objective from its `LPex1.java` counterpart. Here the objective function is created and added to the model like this:

```
     // Q = 0.5 ( 33*x0*x0 + 22*x1*x1 + 11*x2*x2 - 12*x0*x1 - 23*x1*x2 )
          IloNumExpr x00 = model.prod( 33, x[0], x[0]);
          IloNumExpr x11 = model.prod( 22, x[1], x[1]);
          IloNumExpr x22 = model.prod( 11, x[2], x[2]);
          IloNumExpr x01 = model.prod(-12, x[0], x[1]);
          IloNumExpr x12 = model.prod(-23, x[1], x[2]);
          IloNumExpr Q   = model.prod(0.5, model.sum(x00, x11, x22, x01, x12));

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

A quadratic objective may be built with `square`, `prod` or `sum` methods. Note that inclusion of `IloPiecewiseLinear` will change the model from a QP to a MIQP.

Complete Program: QPex1.java

```
// --------------------------------------------------------------------------
// File: examples/src/QPex1.java
// Version 9.0
// --------------------------------------------------------------------------
//  Copyright (C) 2001-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// QPex1.java - Entering and optimizing a QP problem

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


public class QPex1 {
   public static void main(String[] args) {
```

```
      try {
         IloCplex cplex = new IloCplex();
         IloLPMatrix lp = populateByRow(cplex);

//#ifdef FULLTEST
         cplex.setParam(IloCplex.IntParam.Threads,
                        cplex.getMax(IloCplex.IntParam.Threads));
//#endif

         if ( cplex.solve() ) {
            double[] x     = cplex.getValues(lp);
            double[] dj    = cplex.getReducedCosts(lp);
            double[] pi    = cplex.getDuals(lp);
            double[] slack = cplex.getSlacks(lp);

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

            int ncols = lp.getNcols();
            for (int j = 0; j < ncols; ++j) {
               System.out.println("Column: " + j +
                                  " Value = " + x[j] +
                                  " Reduced cost = " + dj[j]);
            }

            int nrows = lp.getNrows();
            for (int i = 0; i < nrows; ++i) {
               System.out.println("Row   : " + i +
                                  " Slack = " + slack[i] +
                                  " Pi = " + pi[i]);
            }

            cplex.exportModel("qpex1.lp");
         }
         cplex.end();
      }
      catch (IloException e) {
         System.err.println("Concert exception '" + e + "' caught");
      }
   }

   static IloLPMatrix populateByRow(IloMPModeler model) throws IloException {
      IloLPMatrix lp = model.addLPMatrix();

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

      double[]   lhs = {-Double.MAX_VALUE, -Double.MAX_VALUE};
      double[]   rhs = {20.0, 30.0};
      double[][] val = { {-1.0,  1.0,  1.0},
                         { 1.0, -3.0,  1.0} };
      int[][]    ind = { {0, 1, 2},
                         {0, 1, 2} };
      lp.addRows(lhs, rhs, ind, val);

      // Q = 0.5 ( 33*x0*x0 + 22*x1*x1 + 11*x2*x2 - 12*x0*x1 - 23*x1*x2 )
      IloNumExpr x00 = model.prod( 33, x[0], x[0]);
```

```
      IloNumExpr x11 = model.prod( 22, x[1], x[1]);
      IloNumExpr x22 = model.prod( 11, x[2], x[2]);
      IloNumExpr x01 = model.prod(-12, x[0], x[1]);
      IloNumExpr x12 = model.prod(-23, x[1], x[2]);
      IloNumExpr Q   = model.prod(0.5, model.sum(x00, x11, x22, x01, x12));

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

      return (lp);
   }
}
```

Example: qpex1.c

This example shows you how to optimize a QP with routines from the ILOG CPLEX Callable Library when the problem data is stored in a file. The example derives from `lpex1.c` from the ILOG CPLEX *Getting Started* manual. The Concert versions of this example, `iloqpex1.cpp` and `QPex1.java`, are included online in the standard distribution.

Instead of calling `CPXlpopt` to find a solution as for the *linear* programming problem in `lpex1.c`, this example calls `CPXqpopt` to optimize this *quadratic* programming problem.

Like other applications based on the ILOG CPLEX Callable Library, this one begins with calls to `CPXopenCPLEX` to initialize the ILOG CPLEX environment and to `CPXcreateprob` to create the problem object. Before it ends, it frees the problem object with a call to `CPXfreeprob`, and it frees the environment with a call to `CPXcloseCPLEX`.

In the routine `setproblemdata`, there are parameters for `qmatbeg`, `qmatcnt`, `qmatind`, and `qmatval` to fill the quadratic coefficient matrix. The Callable Library routine `CPXcopyquad` copies this data into the problem object created by the Callable Library routine `CPXcreateprob`.

In this example, the problem is a *maximization*, so the objective sense is specified as `CPX_MAX`.

The off-diagonal terms in the matrix $\boldsymbol{Q}$ are one-half the value of the terms x_1x_2, and x_2x_3 as they appear in the algebraic form of the example.

Instead of calling `CPXlpopt` to find a solution as for the *linear* programming problem in `lpex1.c`, this example calls `CPXqpopt` to optimize this *quadratic* programming problem.

Complete Program: qpex1.c

The complete program, `qpex1.c`, appears here and online in the standard distribution.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/qpex1.c                                              */
/*  Version 9.0                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2003 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
```

```
/*  Permission is expressly granted to use this example in the           */
/*  course of developing applications that use ILOG products.            */
/*---------------------------------------------------------------------*/

/* qpex1.c - Entering and optimizing a quadratic programming problem */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with include of cplex.h. */

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

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

#include <string.h>

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

static int
   setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                   int *objsen_p, double **obj_p, double **rhs_p,
                   char **sense_p, int **matbeg_p, int **matcnt_p,
                   int **matind_p, double **matval_p, double **lb_p,
                   double **ub_p, int **qmatbeg_p, int **qmatcnt_p,
                   int **qmatind_p, double **qmatval_p);

static void
   free_and_null (char **ptr);


/* The problem we are optimizing will have 2 rows, 3 columns,
   6 nonzeros, and 7 nonzeros in the quadratic coefficient matrix. */

#define NUMROWS    2
#define NUMCOLS    3
#define NUMNZ      6
#define NUMQNZ     7


int
main (void)
{
   /* Declare pointers for the variables and arrays that will contain
      the data which define the LP problem.  The setproblemdata() routine
      allocates space for the problem data.  */

   char     *probname = NULL;
   int      numcols;
   int      numrows;
   int      objsen;
   double   *obj = NULL;
   double   *rhs = NULL;
   char     *sense = NULL;
   int      *matbeg = NULL;
   int      *matcnt = NULL;
   int      *matind = NULL;
   double   *matval = NULL;
   double   *lb = NULL;
```

```
double   *ub = NULL;
int      *qmatbeg = NULL;
int      *qmatcnt = NULL;
int      *qmatind = NULL;
double   *qmatval = NULL;

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

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


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

/* Initialize the CPLEX environment */

env = CPXopenCPLEX (&status);

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

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

/* Turn on output to the screen */

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

/* Fill in the data for the problem.  */

status = setproblemdata (&probname, &numcols, &numrows, &objsen, &obj,
                         &rhs, &sense, &matbeg, &matcnt, &matind,
                         &matval, &lb, &ub, &qmatbeg, &qmatcnt,
```

```
                              &qmatind, &qmatval);
   if ( status ) {
      fprintf (stderr, "Failed to build problem data arrays.\n");
      goto TERMINATE;
   }

   /* Create the problem. */

   lp = CPXcreateprob (env, &status, probname);

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

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

   /* Now copy the LP part of the problem data into the lp */

   status = CPXcopylp (env, lp, numcols, numrows, objsen, obj, rhs,
                       sense, matbeg, matcnt, matind, matval,
                       lb, ub, NULL);

   if ( status ) {
      fprintf (stderr, "Failed to copy problem data.\n");
      goto TERMINATE;
   }

   status = CPXcopyquad (env, lp, qmatbeg, qmatcnt, qmatind, qmatval);
   if ( status ) {
      fprintf (stderr, "Failed to copy quadratic matrix.\n");
      goto TERMINATE;
   }


   /* Optimize the problem and obtain solution. */

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

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


   /* Write the output to the screen. */

   printf ("\nSolution status = %d\n", solstat);
```

```
   printf ("Solution value  = %f\n\n", objval);

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

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

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

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

   status = CPXwriteprob (env, lp, "qpex1.lp", NULL);
   if ( status ) {
      fprintf (stderr, "Failed to write LP to disk.\n");
      goto TERMINATE;
   }

TERMINATE:

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

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

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

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

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

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

   /* Free up the problem data arrays, if necessary. */

   free_and_null ((char **) &probname);
```

```
   free_and_null ((char **) &obj);
   free_and_null ((char **) &rhs);
   free_and_null ((char **) &sense);
   free_and_null ((char **) &matbeg);
   free_and_null ((char **) &matcnt);
   free_and_null ((char **) &matind);
   free_and_null ((char **) &matval);
   free_and_null ((char **) &lb);
   free_and_null ((char **) &ub);
   free_and_null ((char **) &qmatbeg);
   free_and_null ((char **) &qmatcnt);
   free_and_null ((char **) &qmatind);
   free_and_null ((char **) &qmatval);

   return (status);

}  /* END main */


/* This function fills in the data structures for the quadratic program:

      Maximize
       obj: x1 + 2 x2 + 3 x3
              - 0.5 ( 33x1*x1 + 22*x2*x2 + 11*x3*x3
                    -  12*x1*x2 - 23*x2*x3 )
      Subject To
       c1: - x1 + x2 + x3 <= 20
       c2: x1 - 3 x2 + x3 <= 30
      Bounds
       0 <= x1 <= 40
      End
 */


static int
setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                int *objsen_p, double **obj_p, double **rhs_p,
                char **sense_p, int **matbeg_p, int **matcnt_p,
                int **matind_p, double **matval_p, double **lb_p,
                double **ub_p, int **qmatbeg_p, int **qmatcnt_p,
                int **qmatind_p, double **qmatval_p)
{
   char     *zprobname = NULL;     /* Problem name <= 16 characters */
   double   *zobj = NULL;
   double   *zrhs = NULL;
   char     *zsense = NULL;
   int      *zmatbeg = NULL;
   int      *zmatcnt = NULL;
   int      *zmatind = NULL;
   double   *zmatval = NULL;
   double   *zlb = NULL;
   double   *zub = NULL;
   int      *zqmatbeg = NULL;
   int      *zqmatcnt = NULL;
   int      *zqmatind = NULL;
   double   *zqmatval = NULL;
   int      status = 0;
```

```
zprobname = (char *) malloc (16 * sizeof(char));
zobj      = (double *) malloc (NUMCOLS * sizeof(double));
zrhs      = (double *) malloc (NUMROWS * sizeof(double));
zsense    = (char *) malloc (NUMROWS * sizeof(char));
zmatbeg   = (int *) malloc (NUMCOLS * sizeof(int));
zmatcnt   = (int *) malloc (NUMCOLS * sizeof(int));
zmatind   = (int *) malloc (NUMNZ * sizeof(int));
zmatval   = (double *) malloc (NUMNZ * sizeof(double));
zlb       = (double *) malloc (NUMCOLS * sizeof(double));
zub       = (double *) malloc (NUMCOLS * sizeof(double));
zqmatbeg  = (int *) malloc (NUMCOLS * sizeof(int));
zqmatcnt  = (int *) malloc (NUMCOLS * sizeof(int));
zqmatind  = (int *) malloc (NUMQNZ * sizeof(int));
zqmatval  = (double *) malloc (NUMQNZ * sizeof(double));

if ( zprobname == NULL || zobj     == NULL ||
     zrhs      == NULL || zsense   == NULL ||
     zmatbeg   == NULL || zmatcnt  == NULL ||
     zmatind   == NULL || zmatval  == NULL ||
     zlb       == NULL || zub      == NULL ||
     zqmatbeg  == NULL || zqmatcnt == NULL ||
     zqmatind  == NULL || zqmatval == NULL  )  {
   status = 1;
   goto TERMINATE;
}

strcpy (zprobname, "example");

/* The code is formatted to make a visual correspondence
   between the mathematical linear program and the specific data
   items.   */

  zobj[0]  = 1.0;   zobj[1]   = 2.0;   zobj[2] = 3.0;

zmatbeg[0] = 0;     zmatbeg[1] = 2;    zmatbeg[2] = 4;
zmatcnt[0] = 2;     zmatcnt[1] = 2;    zmatcnt[2] = 2;

zmatind[0] = 0;     zmatind[2] = 0;    zmatind[4] = 0;     zsense[0] = 'L';
zmatval[0] = -1.0;  zmatval[2] = 1.0;  zmatval[4] = 1.0;   zrhs[0]   = 20.0;

zmatind[1] = 1;     zmatind[3] = 1;    zmatind[5] = 1;     zsense[1] = 'L';
zmatval[1] = 1.0;   zmatval[3] = -3.0; zmatval[5] = 1.0;   zrhs[1]   = 30.0;

    zlb[0] = 0.0;       zlb[1] = 0.0;          zlb[2] = 0.0;
    zub[0] = 40.0;      zub[1] = CPX_INFBOUND; zub[2] = CPX_INFBOUND;

/* Now set up the Q matrix.  Note that we set the values knowing that
 * we're doing a maximization problem, so negative values go on
 * the diagonal.  Also, the off diagonal terms are each repeated,
 * by taking the algebraic term and dividing by 2 */

zqmatbeg[0] = 0;     zqmatbeg[1] = 2;     zqmatbeg[2] = 5;
zqmatcnt[0] = 2;     zqmatcnt[1] = 3;     zqmatcnt[2] = 2;

/* Matrix is set up visually.  Note that the x1*x3 term is 0, and is
 * left out of the matrix.  */

zqmatind[0] = 0;     zqmatind[2] = 0;
```

```
   zqmatval[0] = -33.0; zqmatval[2] = 6.0;

   zqmatind[1] = 1;     zqmatind[3] = 1;     zqmatind[5] = 1;
   zqmatval[1] = 6.0;   zqmatval[3] = -22.0; zqmatval[5] = 11.5;

                        zqmatind[4] = 2;     zqmatind[6] = 2;
                        zqmatval[4] = 11.5;  zqmatval[6] = -11.0;

TERMINATE:

   if ( status ) {
      free_and_null ((char **) &zprobname);
      free_and_null ((char **) &zobj);
      free_and_null ((char **) &zrhs);
      free_and_null ((char **) &zsense);
      free_and_null ((char **) &zmatbeg);
      free_and_null ((char **) &zmatcnt);
      free_and_null ((char **) &zmatind);
      free_and_null ((char **) &zmatval);
      free_and_null ((char **) &zlb);
      free_and_null ((char **) &zub);
      free_and_null ((char **) &zqmatbeg);
      free_and_null ((char **) &zqmatcnt);
      free_and_null ((char **) &zqmatind);
      free_and_null ((char **) &zqmatval);
   }
   else {
      *numcols_p   = NUMCOLS;
      *numrows_p   = NUMROWS;
      *objsen_p    = CPX_MAX;   /* The problem is maximization */

      *probname_p  = zprobname;
      *obj_p       = zobj;
      *rhs_p       = zrhs;
      *sense_p     = zsense;
      *matbeg_p    = zmatbeg;
      *matcnt_p    = zmatcnt;
      *matind_p    = zmatind;
      *matval_p    = zmatval;
      *lb_p        = zlb;
      *ub_p        = zub;
      *qmatbeg_p   = zqmatbeg;
      *qmatcnt_p   = zqmatcnt;
      *qmatind_p   = zqmatind;
      *qmatval_p   = zqmatval;
   }
   return (status);

}  /* END setproblemdata */


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

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
```

```
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */
```

Example: Reading a QP from a File qpex2.c

This example shows you how to optimize a QP with routines from the ILOG CPLEX Callable Library when the problem data is stored in a file. The example derives from `lpex2.c` from the ILOG CPLEX *Getting Started* manual. The Concert versions of this example, `iloqpex2.cpp` and `QPex2.java`, are included online in the standard distribution.

Instead of calling `CPXlpopt` to find a solution as for the *linear* programming problem in `lpeq2.c`, this example calls `CPXqpopt` to optimize this *quadratic* programming problem.

Like other applications based on the ILOG CPLEX Callable Library, this one begins with calls to `CPXopenCPLEX` to initialize the ILOG CPLEX environment and to `CPXcreateprob` to create the problem object. Before it ends, it frees the problem object with a call to `CPXfreeprob`, and it frees the environment with a call to `CPXcloseCPLEX`.

The complete program, `qpex2.c`, appears here and online in the standard distribution.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/qpex2.c                                              */
/*  Version 9.0                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2003 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* qpex2.c - Reading in and optimizing a QP problem */

/* To run this example, command line arguments are required.
   i.e.,   qpex2   filename   method
   where
       filename is the name of the file, with .mps, .lp, or .sav extension
       method   is the optimization method
                   o          default
                   p          primal simplex
                   d          dual   simplex
                   n          network with dual simplex cleanup
                   b          barrier without crossover
   Example:
       qpex2  example.mps  o
 */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

/* Bring in the declarations for the string and character functions
   and malloc */

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

/* Include declarations for functions in this program */

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


int
main (int argc, char *argv[])
{
   /* Declare and allocate space for the variables and arrays where we will
      store the optimization results including the status, objective value,
      maximum bound violation, variable values, and basis. */

   int      solnstat, solnmethod, solntype;
   double   objval, maxviol;
   double   *x     = NULL;
   int      *cstat = NULL;
   int      *rstat = NULL;
```

```
CPXENVptr     env = NULL;
CPXLPptr      lp = NULL;
int           status = 0;
int           j;
int           cur_numrows, cur_numcols;
int           method;

char          *basismsg;

/* Check the command line arguments */

if (( argc != 3 )                          ||
    ( strchr ("podbn", argv[2][0]) == NULL )  ) {
   usage (argv[0]);
   goto TERMINATE;
}

/* Initialize the CPLEX environment */

env = CPXopenCPLEX (&status);

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

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

/* Turn on output to the screen */

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

/* Create the problem, using the filename as the problem name */

lp = CPXcreateprob (env, &status, argv[1]);

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

if ( lp == NULL ) {
```

```
   fprintf (stderr, "Failed to create LP.\n");
   goto TERMINATE;
}

/* Now read the file, and copy the data into the created lp */

status = CPXreadcopyprob (env, lp, argv[1], NULL);
if ( status ) {
   fprintf (stderr, "Failed to read and copy the problem data.\n");
   goto TERMINATE;
}

if ( CPXgetprobtype (env, lp) != CPXPROB_QP ) {
   fprintf (stderr, "Input file is not a QP.  Exiting.\n");
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */

switch (argv[2][0]) {
   case 'o':
      method = CPX_ALG_AUTOMATIC;
      break;
   case 'p':
      method = CPX_ALG_PRIMAL;
      break;
   case 'd':
      method = CPX_ALG_DUAL;
      break;
   case 'n':
      method = CPX_ALG_NET;
      break;
   case 'b':
      method = CPX_ALG_BARRIER;
      break;
   default:
      method = CPX_ALG_NONE;
      break;
}

status = CPXsetintparam (env, CPX_PARAM_QPMETHOD, method);
if ( status ) {
   fprintf (stderr,
            "Failed to set the optimization method, error %d.\n", status);
   goto TERMINATE;
}

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

solnstat = CPXgetstat (env, lp);

if      ( solnstat == CPX_STAT_UNBOUNDED ) {
   printf ("Model is unbounded\n");
   goto TERMINATE;
```

```
}
else if ( solnstat == CPX_STAT_INFEASIBLE ) {
   printf ("Model is infeasible\n");
   goto TERMINATE;
}
else if ( solnstat == CPX_STAT_INFOrUNBD ) {
   printf ("Model is infeasible or unbounded\n");
   goto TERMINATE;
}

status = CPXsolninfo (env, lp, &solnmethod, &solntype, NULL, NULL);
if ( status ) {
   fprintf (stderr, "Failed to obtain solution info.\n");
   goto TERMINATE;
}
printf ("Solution status %d, solution method %d\n", solnstat, solnmethod);

if ( solntype == CPX_NO_SOLN ) {
   fprintf (stderr, "Solution not available.\n");
   goto TERMINATE;
}

status = CPXgetobjval (env, lp, &objval);
if ( status ) {
   fprintf (stderr, "Failed to obtain objective value.\n");
   goto TERMINATE;
}
printf ("Objective value %.10g.\n", objval);

/* The size of the problem should be obtained by asking CPLEX what
   the actual size is.  cur_numrows and cur_numcols store the
   current number of rows and columns, respectively.  */

cur_numcols = CPXgetnumcols (env, lp);
cur_numrows = CPXgetnumrows (env, lp);

/* Retrieve basis, if one is available */

if ( solntype == CPX_BASIC_SOLN ) {
   cstat = (int *) malloc (cur_numcols*sizeof(int));
   rstat = (int *) malloc (cur_numrows*sizeof(int));
   if ( cstat == NULL || rstat == NULL ) {
      fprintf (stderr, "No memory for basis statuses.\n");
      goto TERMINATE;
   }

   status = CPXgetbase (env, lp, cstat, rstat);
   if ( status ) {
      fprintf (stderr, "Failed to get basis; error %d.\n", status);
      goto TERMINATE;
   }
}
else {
   printf ("No basis available\n");
}


/* Retrieve solution vector */
```

```
   x = (double *) malloc (cur_numcols*sizeof(double));
   if ( x == NULL ) {
      fprintf (stderr, "No memory for solution.\n");
      goto TERMINATE;
   }

   status = CPXgetx (env, lp, x, 0, cur_numcols-1);
   if ( status ) {
      fprintf (stderr, "Failed to obtain primal solution.\n");
      goto TERMINATE;
   }


   /* Write out the solution */

   for (j = 0; j < cur_numcols; j++) {
      printf ( "Column %d:  Value = %17.10g", j, x[j]);
      if ( cstat != NULL ) {
         switch (cstat[j]) {
            case CPX_AT_LOWER:
               basismsg = "Nonbasic at lower bound";
               break;
            case CPX_BASIC:
               basismsg = "Basic";
               break;
            case CPX_AT_UPPER:
               basismsg = "Nonbasic at upper bound";
               break;
            case CPX_FREE_SUPER:
               basismsg = "Superbasic, or free variable at zero";
               break;
            default:
               basismsg = "Bad basis status";
               break;
         }
         printf ("  %s",basismsg);
      }
      printf ("\n");
   }

   /* Display the maximum bound violation. */

   status = CPXgetdblquality (env, lp, &maxviol, CPX_MAX_PRIMAL_INFEAS);
   if ( status ) {
      fprintf (stderr, "Failed to obtain bound violation.\n");
      goto TERMINATE;
   }
   printf ("Maximum bound violation = %17.10g\n", maxviol);

TERMINATE:

   /* Free up the basis and solution */

   free_and_null ((char **) &cstat);
   free_and_null ((char **) &rstat);
   free_and_null ((char **) &x);
```

```
   /* Free up the problem, if necessary */

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

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

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

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

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

   return (status);

}  /* END main */


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

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */


static void
usage (char *progname)
{
   fprintf (stderr,"Usage: %s filename algorithm\n", progname);
   fprintf (stderr,"   where filename is a file with extension \n");
   fprintf (stderr,"      MPS, SAV, or LP (lower case is allowed)\n");
   fprintf (stderr,"   and algorithm is one of the letters\n");
   fprintf (stderr,"      o          default\n");
   fprintf (stderr,"      p          primal simplex\n");
   fprintf (stderr,"      d          dual simplex\n");
   fprintf (stderr,"      n          network simplex\n");
   fprintf (stderr,"      b          barrier\n");
   fprintf (stderr," Exiting...\n");
} /* END usage */
```

C H A P T E R

12

Solving Problems with Quadratic Constraints (QCP)

This chapter tells you how to solve *quadratically constrained* programming problems (QCPs). This chapter contains sections about:

- *Identifying a Quadratically Constrained Program (QCP)* on page 296;
- *Determining Problem Type* on page 299;
- *Changing Problem Type* on page 299
- *Changing Quadratic Constraints* on page 300;
- *Solving with Quadratic Constraints* on page 300;
- *Numerical Difficulties and Quadratic Constraints* on page 301.

Identifying a Quadratically Constrained Program (QCP)

The distinguishing characteristic of QCP is that quadratic terms may appear in one or more constraints of the problem. The objective function of such a problem may or may not contain quadratic terms as well. Thus, the most general formulation of a QCP is:

Minimize $\frac{1}{2}\boldsymbol{x}^{\mathrm{T}}\boldsymbol{Q}x + \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}$

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

and $a_i^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{x}^{\mathrm{T}}\boldsymbol{Q}_i\boldsymbol{x} \le r_i$ for $i=1,\ldots,q$

with these bounds $l \le \boldsymbol{x} \le u$

As with a quadratic objective function, convexity plays an important role in quadratic constraints. The constraints must each define a convex region. To make sure of convexity, ILOG CPLEX requires that each $\boldsymbol{Q_i}$ matrix be positive semi-definite (PSD). The following sections offer more information about these concepts.

Convexity

The inequality $x^2 + y^2 \le 1$ is convex. To give you an intuitive idea about convexity, Figure 12.1 graphs that inequality and shades the area that it defines as a constraint. If you consider *a* and *b* as arbitrary values in the domain of the constraint, you see that any continuous line segment between them is contained entirely in the domain.

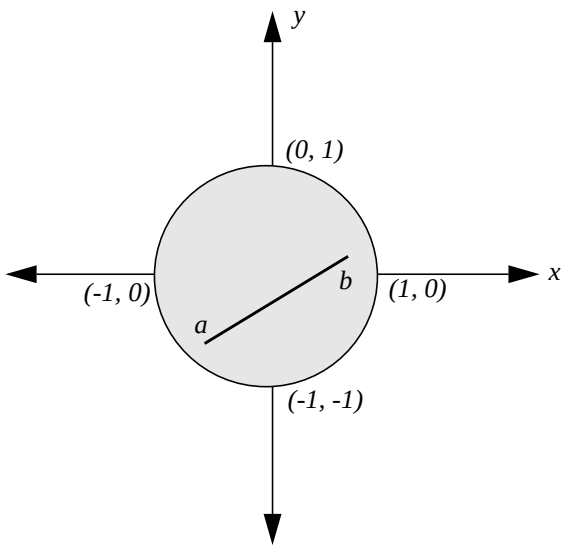

Figure 12.1 *$x^2 + y^2 \le 1$ is convex*

The inequality $x^2 + y^2 \geq 1$ is not convex; it is concave. Figure 12.2 graphs that inequality and shades the area that it defines as a constraint. If you consider *c* and *d* as arbitrary values in the domain of this constraint, then you see that there may be continuous line segments that join the two values in the domain but pass outside the domain of the constraint to do so.

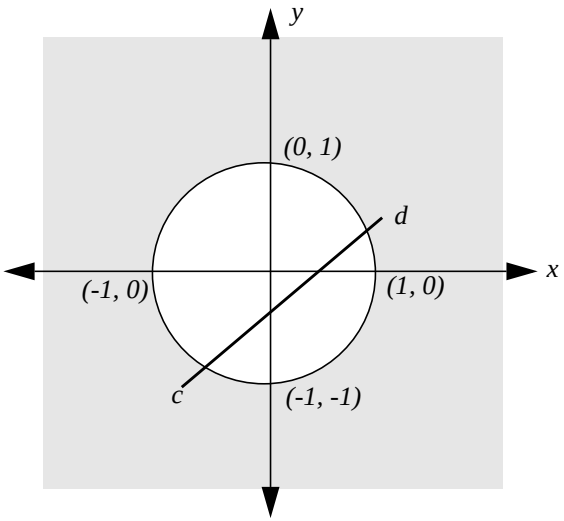

Figure 12.2 *$x^2 + y^2 \geq 1$ is not convex*

It might be less obvious at first glance that the equality $x^2 + y^2 = 1$ is not convex either. As you see in Figure 12.3, there may be a continuous line segment that joins two arbitrary points, such as *e* and *f*, in the domain but the line segment may pass outside the domain. Another way to see this idea is to note that an equality constraint is algebraically equivalent to the intersection of two inequality constraints of opposite sense, and you have already seen that at least one of those quadratic inequalities will not be convex. Thus, the equality is not convex either.

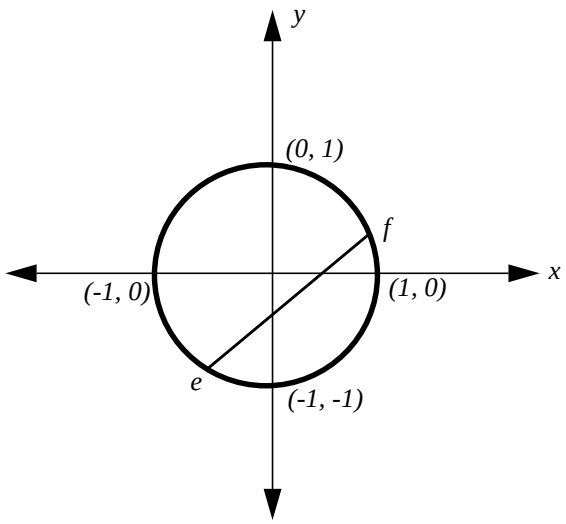

Figure 12.3 *$x^2 + y^2 = 1$ is not convex*

Semi-definiteness

Identifying a Quadratically Constrained Program (QCP) on page 296 explained that the quadratic matrix in each constraint must be positive semi-definite (PSD), thus providing convexity. A matrix $\boldsymbol{Q_i}$ is PSD if $\boldsymbol{x}^T\boldsymbol{Q_i}\boldsymbol{x} \geq 0$ for every vector $\boldsymbol{x}$, whether or not $\boldsymbol{x}$ is feasible. Other issues pertaining to positive semi-definiteness are discussed in the context of a quadratic objective function in *Identifying Convex QPs* on page 266.

When you call the barrier optimizer, your quadratic constraints will be checked for the necessary PSD property, and an error status 5002 will be returned if any of them violate it.

There is one exception to the PSD requirement; that is, there is an additional form of quadratic constraint which is accepted but is not covered by the general formulation in *Identifying a Quadratically Constrained Program (QCP)* on page 296. Technically, the quadratically constrained problem class that the barrier optimizer solves is a Second-Order Cone Program (SOCP). ILOG CPLEX, through its preprocessing feature, makes the translation to SOCP for you, transparently, returning the solution in terms of your original formulation. A constraint will be accepted for solution by the barrier optimizer if it can be transformed to the following convex cone constraint:

$$-x_0^2 + \sum x_i^2 \leq 0$$

That formulation is distinguished primarily by the specific signs of the coefficients and by the lack of a linear term, where x_0 is a nonnegative variable

Determining Problem Type

Concert Technology treats all models as capable of containing quadratic constraints. In other words, applications written in Concert Technology are capable of handling quadratic constraints. These constraints can be added or deleted at will in your application. When extracting a model with a quadratic constraint, `IloCplex` will automatically detect it as a QCP and make the required adjustments to its internal data structures.

When routines of the Callable Library read a problem from a file, they are capable of detecting quadratic constraints. If they detect a quadratic constraint in the model they read, Callable Library routines will automatically set the problem type as QCP. If there are no quadratic constraints, then Callable Library routines consider whether there are any quadratic coefficients in the objective function. If there is a quadratic term in the objective function, then Callable Library routines automatically set the problem type as QP, as explained in *Changing Problem Type in QPs* on page 269.

In the Interactive Optimizer, a problem containing a quadratic constraint, as denoted by square brackets, is automatically identified as QCP when the problem is read from a file or entered interactively.

ILOG CPLEX supports the definition of quadratic constraints in `.sav` files and in `.lp` files. In `.lp` files, you can state your quadratic constraints in the **subject to** section of the file.

Changing Problem Type

By default, every model in Concert Technology is the most general problem type possible. Consequently, it is not necessary to declare the problem type nor to change the problem type, even if you add quadratic constraints to the model or remove them from it. In contrast, both the Callable Library and the Interactive Optimizer need for you to indicate a change in problem type explicitly if you remove the quadratic constraints that make your model a QCP.

In both the Callable Library and Interactive Optimizer, if you want to remove the quadratic constraints in order to solve the problem as an LP or a QP, then you must first change the problem type, just as you would, for example, if you removed the quadratic coefficients from a quadratic objective function.

From the Callable Library, use the routine `CPXchgprobtype` to change the problem type to `CPXPROB_LP` if you remove the quadratic constraints from your model in order to solve it as an LP. Contrariwise, if you want to add quadratic constraints to an LP or a QP model and then solve it as a QCP, use the routine `CPXchgprobtype` to change the problem type to `CPXPROB_QCP`.

When using the Interactive Optimizer, you apply the command `change problem` with one of the following options:

- `lp` indicates that you want ILOG CPLEX to treat the problem as an LP. This change in the problem type removes all the quadratic information from your problem, if there is any present.
- `qp` indicates that you want ILOG CPLEX to treat the problem as a QP (that is, a problem with a quadratic objective). This choice removes the quadratic constraints, if there were any in the model.
- `qcp` indicates that you want ILOG CPLEX to treat the problem as a QCP.

Changing Quadratic Constraints

To modify a quadratic constraint in your model, you must first delete the old quadratic constraint and then add the new one.

In Concert Technology, you add constraints (whether or not they are quadratic) by means of the method `add` of the class `IloModel`, as explained about C++ applications in *Adding Constraints: IloConstraint and IloRange* on page 46 and about Java applications in *The Active Model* on page 79. To add constraints to a model in the .NET framework, see *ILOG Concert Technology for C#.NET Users* on page 101.

Also in Concert Technology, you can remove constraints (again, whether or not they are quadratic) by means of the method `remove` of the class `IloModel`, as explained about C++ applications in *Deleting and Removing Modeling Objects* on page 58 and about Java applications in *Modifying the Model* on page 98.

The Callable Library has a separate set of routines for creating and modifying quadratic constraints; do **not** use the routines that create or modify linear constraints.

In the Callable Library, you add a quadratic constraint by means of the routine `CPXaddqconstr`. You remove and delete quadratic constraints by means of the routine `CPXdelqconstr`. Don't forget to change the problem type, as explained in *Changing Problem Type* on page 299. If you want to change a quadratic constraint, first delete it by calling `CPXdelqconstrs` and then add the new constraint using `CPXaddqconstr`.

In the Interactive Optimizer, if you want to change a quadratic constraint, you must delete the constraint (`change delete qconstraints`) and add the new constraint. Again, you must change the problem type, as explained in *Changing Problem Type* on page 299.

Solving with Quadratic Constraints

ILOG CPLEX allows you to solve your QCP models (that is, problems with quadratic constraints) through a simple interface, by calling the default optimizer as follows:

- In Concert applications, use the `solve` method of `IloCplex`.

- From the Callable Library, use the routine `CPXbaropt`.
- In the Interactive Optimizer, use the command `optimize`.

With default settings, each of these approaches will result in the barrier optimizer being called to solve a continuous QCP.

The barrier optimizer is the only optimizer available to solve QCPs.

Numerical Difficulties and Quadratic Constraints

A word of warning: numerical difficulties are likely to be more acute for QCP than for LP or QP. Consequently, you will need to scale your variables carefully so that units of measure are roughly comparable among them.

Examples: QCP

For examples of QCPs, see these variations of the same problem in *yourCPLEXhome*`/examples`:

- `qcpex1.c`
- `iloqcpex1.cpp`
- `QCPex1.java`
- `QCPex1.cs`

Part IV

Discrete Optimization

This part focuses on algorithmic considerations about the ILOG CPLEX optimizers that solve problems formulated in terms of **discrete** variables, such as integer, Boolean, piecewise-linear, or semi-continuous variables. While default settings of ILOG CPLEX enable you to solve many problems without changing parameters, this part also documents features that enable you to tune performance. This part contains:

- *Solving Mixed Integer Programming Problems (MIP)* on page 305
- *Cutting Stock: Column Generation* on page 375
- *Rates: Using Semi-Continuous Variables* on page 389
- *Transport: Piecewise Linear Optimization* on page 397
- *Logical Constraints in Optimization* on page 411
- *Food Manufacturing 2: Using Logical Constraints* on page 419
- *Early Tardy Scheduling* on page 429

C H A P T E R

13

Solving Mixed Integer Programming Problems (MIP)

The ILOG CPLEX *Mixed Integer Optimizer* enables you to solve models in which one or more variables must take integer solution values. This chapter tells you more about optimizing mixed integer programming (MIP) problems with ILOG CPLEX. It includes information about:

- *Stating a MIP Problem* on page 306
- *Considering Preliminary Issues* on page 307
- *Changing Problem Type in MIPs* on page 309
- *Using the Mixed Integer Optimizer* on page 311
- *Using the MIP Solution* on page 327
- *Using Special Ordered Sets* on page 329
- *Using Semi-Continuous Variables* on page 331
- *Progress Reports: Interpreting the Node Log* on page 331
- *Troubleshooting MIP Performance Problems* on page 335
- *Example: Optimizing a Basic MIP Problem* on page 346
- *Example: Reading a MIP Problem from a File* on page 356

- *Example: Using SOS and Priority* on page 362

Stating a MIP Problem

A mixed integer programming (MIP) problem may contain both integer and continuous variables. If the problem contains an objective function with no quadratic term, (a *linear objective)*, then the problem is termed a *Mixed Integer Linear Program* (MILP). If there is a quadratic term in the objective function, the problem is termed a *Mixed Integer Quadratic Program* (MIQP). If the model has any constraints containing a quadratic term, regardless of the objective function, the problem is termed a *Mixed Integer Quadratically Constrained Program* (MIQCP).

In ILOG CPLEX documentation, if the discussion pertains specifically to either the MILP or MIQP case, then that term is used. For the majority of topics that pertain equally to MILP, MIQP, and MIQCP, the comprehensive term MIP is used.

Integer variables may be restricted to the values 0 (zero) and 1 (one), in which case they are referred to as *binary* variables. Or they may take on any integer values, in which case they are referred to as *general* integer variables. A variable of any MIP that may take either the value 0 (zero) or a value between a lower and an upper bound is referred to as *semi-continuous*. A semi-continuous variable that is restricted to integer values is referred to as *semi-integer*. *Using Semi-Continuous Variables* on page 331 says a bit more about semi-continuous variables later in this chapter. Special Ordered Sets (SOS) are discussed in *Using Special Ordered Sets* on page 329. Continuous variables in a MIP problem are those which are *not* restricted in any of these ways, and are thus permitted to take any solution value within their (possibly infinite) lower and upper bounds.

In ILOG CPLEX documentation, the comprehensive term *integer variable* means any of the various types just mentioned except for continuous or SOS. The presence or absence of a quadratic term in the objective function or among the constraints for a given variable has no bearing on its being classified as continuous or integer.

The following illustrates a mixed integer programming problem, which is solved in the example program `ilomipex1.cpp` / `mipex1.c`, discussed later in this chapter:

$$
\begin{array}{lrrrrrrrrrr}
\text{Maximize} & & x_1 & + & 2x_2 & + & 3x_3 & + & x_4 & & \\
\text{subject to} & - & x_1 & + & x_2 & + & x_3 & + & 10x_4 & \leq & 20 \\
 & & x_1 & - & 3x_2 & + & x_3 & & & \leq & 30 \\
 & & & & x_2 & & & - & 3.5x_4 & = & 0 \\
\text{with these bounds} & & 0 & \leq & x_1 & \leq & 40 & & & & \\
 & & 0 & \leq & x_2 & \leq & +\infty & & & & \\
 & & 0 & \leq & x_3 & \leq & +\infty & & & &
\end{array}
$$

$$2 \leq x_4 \leq 3$$
$$x_4 \quad \textit{integer}$$

Considering Preliminary Issues

When you are optimizing a MIP, there are a few preliminary issues that you need to consider to get the most out of ILOG CPLEX. The following sections cover such topics as entering variable type, displaying MIPs in the Interactive Optimizer, determining the problem type, and switching to fixed versions of your problem.

- *Entering MIP Problems* on page 307
- *Displaying MIP Problems* on page 308

Entering MIP Problems

You enter MIPs into ILOG CPLEX as explained in each of the chapters about the APIs of ILOG CPLEX, with this additional consideration: you need to indicate which variables are binary, general integer, semi-continuous, and semi-integer, and which are contained in special ordered sets (SOS).

Concert Technology users can specify this information by passing the value of a type to the appropriate constructor when creating the variable, as summarized in Table 13.1.

Table 13.1 *Specifying Type of Variable in a MIP*

Type of Variable	C++ API	Java API	C#.NET API
binary	`IloNumVar::Type::ILOBOOL`	`IloNumVarType.Bool`	`NumVarType.Bool`
integer	`IloNumVar::Type::ILOINT`	`IloNumVarType.Int`	`NumVarType.Int`
semi-continuous	`IloSemiContVar::Type::ILONUM`	`IloNumVarType.Float`	`NumVarType.Float`
semi-integer	`IloSemiContVar::Type::ILOINT`	`IloNumVarType.Int`	`NumVarType.Int`

Callable Library users can specify this information through the routine `CPXcopyctype`.

In the Interactive Optimizer, to indicate binary integers in the context of the `enter` command, type `binaries` on a separate line, followed by the designated binary variables. To indicate general integers, type `generals` on a separate line, followed by the designated general variables. To indicate semi-continuous variables, type `semi-continuous` on a separate line, followed by the designated variables. Semi-integer variables are indicated by being specified as **both** general integer and semi-continuous. The order of these three

sections does not matter. To enter the general integer variable of the *Stating a MIP Problem* on page 306, you type this:

```
generals
x4
```

You may also read MIP data in from a formatted file, just as you do for linear programming problems. *Understanding File Formats* on page 156 in this manual lists the file formats briefly, and the *ILOG CPLEX Reference Manual* documents file formats, such as MPS, LP, and others.

- To read MIP problem data into the Interactive Optimizer, use the `read` command with an option to indicate the file type.
- To read MIP problem data into your application, use the `importModel` method in Concert Technology or use `CPXreadcopyprob` in the Callable Library.

Displaying MIP Problems

Table 13.2 summarizes display options in the Interactive Optimizer that are specific to MIP problems.

Table 13.2 *Interactive Optimizer Display Options for MIP Problems*

Interactive command	Purpose
`display problem binaries`	lists variables restricted to binary values
`display problem generals`	lists variables restricted to integer values
`display problem semi-continuous`	lists variables of type semi-continuous and semi-integer
`display problem integers`	lists all of the above
`display problem sos`	lists the names of variables in one or more Special Ordered Sets
`display problem stats`	lists LP statistics plus: • binary variable types, if present; • general variable types, if present; • and number of SOS, if present.

In Concert Technology, use one of the accessors supplied with the appropriate object class, such as `IloSOS2::getVariables`.

From the Callable Library, use the routines `CPXgetctype` and `CPXgetsos` to access this information.

Changing Problem Type in MIPs

Concert Technology applications treat all models as capable of containing integer variables, and thus these variable declarations may be added or deleted at will. When extracting a model with integer variables, it will automatically detect it as a MIP and make the required adjustments to internal data structures.

However, the other ways of using ILOG CPLEX, the Callable Library and the Interactive Optimizer, require an explicit notion of a Problem Type to distinguish continuous LPs, QPs, and QCPs from MIPs. Techniques to determine the Problem Type with the Callable Library and the Interactive Optimizer are discussed in this topic.

When you enter a problem, ILOG CPLEX determines the Problem Type from the available information. If the problem is read from a file (`LP`, `MPS`, or `SAV` format, for example), or entered interactively, the Problem Type is determined according to Table 13.3.

Table 13.3 *Problem Type Definitions*

Problem Type	No Integer Variables	Has Integer Variables	No Quadratic Terms in the Objective Function	Has Quadratic Terms in the Objective Function	Has Quadratic Terms in Constraints
`lp`	X		X		
`qp`	X			X	
`qcp`	X			possibly	X
`milp`		X	X		
`miqp`		X		X	
`miqcp`		X		possibly	X

However, if you enter a problem with no integer variables, so that its Problem Type is initially `lp`, `qp`, or `qcp`, and you then wish to modify the problem to contain integer variables, this is accomplished by first changing the Problem Type to `milp, miqcp,` or `miqp`. Conversely, if you have entered an MILP, MIQCP, or MIQP model and wish to remove all the integer declarations and thus convert the model to a continuous formulation, you can change the Problem Type to `lp`, `qp`, or `qcp`. Note that deleting each of the integer variable declarations individually still leaves the Problem Type as `milp`, `miqp`, or `miqcp`, although in most instances the distinction between this problem and its `lp` counterpart is somewhat arbitrary in terms of the steps that will be taken to solve it.

Thus, when using the Interactive Optimizer, you use the command `change problem` with one of the following options:

- `milp, miqcp,` or `miqp`

  indicating that you want ILOG CPLEX to treat the problem as an MILP, MIQCP, or MIQP, respectively. This change in Problem Type makes the model ready for declaration of the integer variables via subsequent `change type` commands. If you change the problem to be an MIQP and there are not already quadratic terms in the objective function, an empty quadratic matrix is created, ready for populating via the `change qpterm` command.

- `lp, qcp,` or `qp`

  indicating that you want all integer declarations removed from the variables in the problem. If you choose the `qp` problem type and there are not already quadratic terms in the objective function, an empty quadratic matrix is created, ready for populating via the `change qpterm` command.

From the Callable Library, use the routine `CPXchgprobtype` to change the Problem Type to `CPXPROB_MILP, CPXPROB_MIQCP,` or `CPXPROB_MIQP` for the MILP, MIQCP, and MIQP case respectively, and then assign integer declarations to the variables through the `CPXcopyctype` function. Conversely, remove all integer declarations from the problem by using `CPXchgprobtype` with Problem Type `CPXPROB_LP, CPXPROB_QCP,` or `CPXPROB_QP`.

At the end of a MIP optimization, the optimal values for the variables are directly available. However, you may wish to obtain information about the LP, QP, or QCP associated with this optimal solution (for example, to know the reduced costs for the continuous variables of the problem at this solution). To do this, you will want to change the problem to be of type Fixed, either `fixed_milp` for the MILP case or `fixed_miqp` for the MIQP case. The fixed MIP is the continuous problem in which the integer variables are fixed at the values they attained in the best integer solution. After changing the problem type, you can then call any of the continuous optimizers to re-optimize, and then display solution information for the continuous version of the problem. If you then wish to change the problem type back to the associated `milp` or `miqp`, you can do so without loss of information in the model.

Changing Variable Type

The command `change type` adds (or removes) the restriction on a variable that it must be an integer. In the Interactive Optimizer, when you enter the command `change type`, the system prompts you to enter the variable that you want to change, and then it prompts you to enter the type (`c` for continuous, `b` for binary, `i` for general integer, `s` for semi-continuous, `n` for semi-integer).

You can change a variable to binary even if its bounds are not 0 (zero) and 1 (one). However, in such a case, the optimizer will change the bounds to be 0 and 1.

If you change the type of a variable to be semi-continuous or semi-integer, be sure to create both a lower bound and an upper bound for it. These variable types specify that at an optimal solution the value for the variable must be either exactly zero or else be between the lower

and upper bounds (and further subject to the restriction that the value be an integer, in the case of semi-integer variables).

By the way, a problem may be changed to a mixed integer problem, even if all its variables are continuous.

> *Note: It is not required to specify explicit bounds on general integer variables. However, if during the branch and cut algorithm a variable exceeds 2,100,000,000 in magnitude of its solution, an error termination will occur. In practice, it is wise to limit integer variables to values far smaller than the stated limit, or numerical difficulties may occur; trying to enforce the difference between 1,000,000 and 1,000,001 on a finite precision computer might work but could be difficult due to roundoff.*

Using the Mixed Integer Optimizer

The ILOG CPLEX Mixed Integer Optimizer solves MIP models using a very general and robust branch & cut algorithm. While MIP models have the potential to be much more difficult than their continuous LP, QCP, and QP counterparts, it is also the case that large MIP models are routinely solved in many production applications. A great deal of algorithmic development effort has been devoted to establishing default ILOG CPLEX parameter settings that achieve good performance on a wide variety of MIP models. Therefore, it is recommended to try solving your model by first calling the Mixed Integer Optimizer in its most straightforward form.

To invoke the Mixed Integer Optimizer:

- In the Interactive Optimizer, use the `mipopt` command.
- In Concert Technology, with the `IloCplex` method `solve`.
- In the Callable Library, use the `CPXmipopt` routine.

Feasibility and Optimality

The following section, *Performance Features of the Mixed Integer Optimizer*, goes into great detail about the algorithmic features, controlled by parameter settings, that are available in ILOG CPLEX to achieve performance tuning on difficult MIP models. However, there is an important parameter, `MIPEmphasis`, that is oriented less toward the user understanding the algorithm being used to solve the model, and more toward the user telling the algorithm something about the underlying aim of the optimization being run. That parameter is discussed here.

Optimizing a MIP model involves:

1. finding a succession of improving integer feasible solutions (solutions satisfying the linear and quadratic constraints and the integrality conditions); while
2. also working toward a proof that no better feasible solution exists and is undiscovered.

For most models, a balance between these two sometimes-competing aims works well, and this is another way of stating the philosophy behind the default `MIPEmphasis` setting: "balance optimality and integer feasibility".

At this default `MIPEmphasis` setting of `0` (that is, `MIPEmphasisBalanced` or `CPX_MIPEMPHASIS_BALANCED`), ILOG CPLEX uses tactics intended to find a proven optimal solution quickly, for models of a broad range of difficulty. That is, considerable analysis of the model is performed before branching ever begins, in the expectation that the investment will result in a faster total run time, yet not every possible analysis is performed. And then branching is performed in a manner that seeks to find good quality feasible solutions, without sacrificing too much time that could be spent proving the optimality of any solution that has already been found.

In many situations, the user will desire a greater emphasis on feasibility and less emphasis on analysis and proof of optimality. For instance, a restrictive time limit (set by the user using the `TiLim` parameter) may be in force due to a real-time application deployment, where a model is of sufficient difficulty that a proof of optimality is unlikely, and the user wants to have simply as good a solution as is practicable when the time limit is reached. The `MIPEmphasis` setting of `1` (`MIPEmphasisFeasibility` or, in the Callable Library, `CPX_MIPEMPHASIS_FEASIBILITY`) directs ILOG CPLEX to adopt this emphasis. Less computational effort is applied at the outset toward the analyses that aid in the eventual proof of optimality, and more effort is spent in immediately beginning computations that search for early (and then improved) feasible solutions. It is likely on most models that an eventual proof of optimality would take longer by setting `MIPEmphasis` to `1`, but since the user has given ILOG CPLEX the additional information that this proof is of less importance than usual, the user's needs will actually be met more effectively.

Another choice for `MIPEmphasis` is `2` (`MIPEmphasisOptimality` or, in the Callable Library, `CPX_MIPEMPHASIS_OPTIMALITY`), which results in a greater emphasis on optimality than on feasibility. The search for feasible solutions is not ignored completely, but the balance is shifted toward moving the Best Bound (described in the following paragraph) more rapidly, at the likely expense of feasible solutions being found less rapidly, and improved feasible solutions less frequently, than under the default emphasis. This setting also provides a general (though not precise) compatibility mode to prior versions of ILOG CPLEX.

The fourth choice for `MIPEmphasis`, `3` (`MIPEmphasisBestBound` or, in the Callable Library, `CPX_MIPEMPHASIS_BESTBOUND`), works exclusively at moving the *Best Bound*. The Best Bound represents the objective function value at which an integer feasible solution could still potentially exist. As possibilities are eliminated, this Best Bound value will move in the opposite direction to that of any improving series of integer feasible solutions. The process of moving the Best Bound will eventually result in the optimal feasible solution

being discovered, at which point the optimization is complete, and feasible solutions may be discovered along the way anyway, due to branches that happen to locate feasible solutions that do not match the Best Bound. A great deal of analysis may be performed on the model, beyond what is done under the default emphasis. Therefore it is recommended to use this setting only on models that are difficult for the default emphasis, and for which you do not care about interim feasible solutions that may or may not be optimal.

The final choice for `MIPEmphasis` is 4 (`CPX_MIPEMPHASIS_HIDDENFEAS`). It applies considerable additional effort toward finding high quality feasible solutions that are difficult to locate, and for this reason the eventual proof of optimality may take longer than with default settings. This choice is intended for use on difficult models where a proof of optimality is unlikely, and where emphasis 1 (one) does not deliver solutions of an appropriately high quality.

To make clear a point that has been alluded to in the above: every choice of `MIPEmphasis` results in the branch & cut algorithm proceeding in a manner that eventually will find and prove an optimal solution, or will prove that no integer feasible solution exists. The choice of emphasis only guides ILOG CPLEX to produce feasible solutions in a way that is in keeping with the user's particular purposes, but the accuracy and completeness of the algorithm is not sacrificed in the process.

The `MIPEmphasis` parameter may be set in conjunction with any other ILOG CPLEX parameters (discussed at length in the next section). For example if you wish to set an upward branching strategy via the `BrDir` parameter, this will be honored under any setting of `MIPEmphasis`. Of course, certain combinations of `MIPEmphasis` with other parameters may be counter-productive, such as turning off all Cuts under emphasis `3`, but the user has the option if that is what is desired.

Performance Features of the Mixed Integer Optimizer

The ILOG CPLEX Mixed Integer Optimizer contains a wealth of features intended to aid in the solution of challenging MIP models. While default strategies are provided that solve the majority of models without user involvement, there exist difficult models that benefit from attention to performance tuning. This section discusses the ILOG CPLEX parameters that are the most likely to offer help on such models.

Because many of these parameter settings directly affect the branch & cut algorithm, here is a description of how that algorithm is implemented within ILOG CPLEX.

Branch & Cut

In the branch & cut algorithm, ILOG CPLEX solves a series of continuous subproblems. To manage those subproblems efficiently, ILOG CPLEX builds a tree in which each subproblem is a node. The root of the tree is the *continuous relaxation* of the original MIP problem.

If the solution to the relaxation has one or more fractional variables, ILOG CPLEX will try to find cuts. Cuts are constraints that cut away areas of the feasible region of the relaxation that contain fractional solutions. ILOG CPLEX can generate several types of cuts. (*Cuts* on page 318 tells you more about that topic.)

If the solution to the relaxation still has one or more fractional-valued integer variables after ILOG CPLEX tries to add cuts, then ILOG CPLEX branches on a fractional variable to generate two new subproblems, each with more restrictive bounds on the branching variable. For example, with binary variables, one node will fix the variable at 0 (zero), the other, at 1 (one).

The subproblems may result in an all-integer solution, in an infeasible solution, or another fractional solution. If the solution is fractional, ILOG CPLEX repeats the process.

ILOG CPLEX cuts off nodes when the value of the objective function associated with the subproblem at that node is worse than the cutoff value.

You set the cutoff value by means of the `CutUp` parameter (for a minimization problem) or the `CutLo` parameter (for a maximization problem), to indicate to ILOG CPLEX that integer feasible solutions worse than this cutoff value should be discarded. The default value of the lower cutoff is `-1e+75` ; the default value of the upper cutoff is `1e+75`. The defaults, in effect, mean that no cutoff is to be supplied. You can supply any number that you find appropriate for your problem. It is never required that you supply a cutoff, and in fact for most applications is it not done.

ILOG CPLEX will use the value of the best integer solution found so far, as modified by the tolerance parameters `ObjDif` or `RelObjDif` (absolute and relative objective function difference, respectively), as the cutoff. Again, it is not typical that users set these parameters, but they are available if you find them useful. Use care in changing these tolerances: if either of them is nonzero, you may miss the optimal solution by as much as that amount. For example, in a model where the true minimum is `100` and the absolute cutoff is set to `5`, if a feasible solution of say, `103` is found at some point, the cutoff will discard all nodes with a solution worse than `98`, and thus the solution of `100` would be overlooked.

Periodically during the branch & cut algorithm, ILOG CPLEX may apply a heuristic process that attempts to compute an integer solution from available information, such as the solution to the relaxation at the current node. This activity does not replace the branching steps, but sometimes is able to inexpensively locate a new feasible solution sooner than by branching, and a solution found in this way is treated in the same way as any other feasible solution. At intervals in the tree, new cuts beyond those computed at the root node may also be added to the problem.

Once ILOG CPLEX finds an integer solution, it does the following:

- it makes that integer solution the *incumbent solution* and that node the *incumbent node*;
- it makes the value of the objective function at that node (modified by the objective difference parameter) the new cutoff value;

- it prunes from the tree all subproblems for which the value of the objective function is no better than the incumbent.

You control the path that CPLEX traverses in the tree through several parameters, as summarized in Table 13.4.

Table 13.4 *Parameters for Controlling Branch & Cut Strategy*

Interactive Optimizer Command	Concert Technology IloCPLEX Method	Callable Library Routine
`set mip strategy backtrack`	`setParam(BtTol, n)`	`CPXsetdblparam(env, CPX_PARAM_BTTOL, n)`
`set mip strategy nodeselect`	`setParam(NodeSel, i)`	`CPXsetintparam(env, CPX_PARAM_NODESEL, i)`
`set mip strategy variableselect`	`setParam(VarSel, i)`	`CPXsetintparam(env, CPX_PARAM_VARSEL, i)`
`set mip strategy bbinterval`	`setParam(BBInterval, i)`	`CPXsetintparam(env, CPX_PARAM_BBINTERVAL,`
`set mip strategy branch`	`setParam(BrDir, i)`	`CPXsetintparam(env, CPX_PARAM_BRDIR, i)`

During the branch & cut algorithm, ILOG CPLEX may choose to continue from the present node and dive deeper into the tree, or it may backtrack (that is, begin a new dive from elsewhere in the tree). The value of the backtrack parameter, `BtTol`, influences this decision, in terms of the relative degradation of the objective function caused by the branches taken so far in this dive. Setting `BtTol` to a value near `0.0` increases the likelihood that a backtrack will occur, while the default value near `1.0` makes it more likely that the present dive will continue to a resolution (fathoming either via a cutoff or an infeasible combination of branches or the discovery of a new incumbent integer feasible solution). See the reference manual *ILOG CPLEX Parameters* for more details about how this parameter influences the computation that determines the decision to backtrack.

When ILOG CPLEX backtracks, there usually remain large numbers of unexplored nodes from which to begin a new dive. The node selection parameter, `NodeSel`, determines this choice.

Table 13.5 *NodeSel Parameter Settings for Node Search Type*

NodeSel Value	Symbolic Value	Node Search Type
`1` (Default)	`CPX_NODESEL_BESTBOUND`	***Best Bound*** search, which means that the node with the best objective function will be selected, generally near the top of the tree.
`2`	`CPX_NODESEL_BESTEST`	***Best Estimate*** search, whereby ILOG CPLEX will estimate a given node's progress toward integer feasibility relative to its degradation of the objective function. This setting can be useful in cases where there is difficulty in finding feasible solutions or in cases where a proof of optimality is not crucial.
`3`	`CPX_NODESEL_BESTEST_ALT`	A variation on the Best Estimate search.
`0`	`CPX_NODESEL_DFS`	***Depth First*** search will be conducted. In many cases this amounts to a brute force strategy for solving the combinatorial problem, gaining a small amount of tactical efficiency due to a variety of reasons, and it is rare that it offers any advantage over other settings.

In instances where Best Estimate node selection (`NodeSel` = `2` or `3`) is in effect, the `BBInterval` parameter determines the frequency at which backtracking is done by Best Bound anyway. The default value of `7` works well, but you can set it to `0` (zero) to make sure that Best Estimate is used every time backtracking occurs.

After a node has been selected, the variable selection parameter, `VarSel`, influences which variable is chosen for branching at that node.

Table 13.6 *VarSel Parameter Settings for Branching Variable Choice*

VarSel Setting	Symbolic Value	Branching Variable Choice
`-1`	`CPX_VARSEL_MININFEAS`	Branch strictly at the nearest integer value which is closest to the fractional variable.
`1`	`CPX_VARSEL_MAXINFEAS`	Branch strictly at the nearest integer value which is furthest from the fractional variable.
`0` (Default)	`CPX_VARSEL_DEFAULT`	ILOG CPLEX makes the determination of each branch direction.
`2`	`CPX_VARSEL_PSEUDO`	Use pseudo costs, which derives an estimate about the effect of each proposed branch from duality information.
`3`	`CPX_VARSEL_STRONG`	Use strong branching, which invests considerable effort in analyzing potential branches in the hope of drastically reducing the number of nodes that will be explored.
`4`	`CPX_VARSEL_PSEUDOREDUCED`	Use pseudo reduced costs, which is a computationally less-intensive version of pseudo costs.

After a variable has been selected for branching, the `BrDir` parameter influences the direction, up or down, of the branch on that variable to be explored first.

Table 13.7 *BrDir Parameter Settings for Branching Direction Choice*

BrDir Setting	Symbolic Value	Branching Direction Choice
`0` (Default)	`CPX_BRANCH_GLOBAL`	ILOG CPLEX makes the determination of each branch direction.
`-1`	`CPX_BRANCH_DOWN`	Branch downward
`1`	`CPX_BRANCH_UP`	Branch upward

Priority orders complement the behavior of these parameters. They are introduced in *Priority Orders* on page 324. They offer a mechanism by which you supply problem-specific directives about the order in which to branch on variables. In a priority order, you can also provide preferred branching directions for specific variables.

On difficult models that generate a great number of nodes in the tree, the amount of available memory for node storage can become a limiting factor. Node files, discussed in *Running Out of Memory* on page 340, can be an effective technique which uses disk space to augment RAM, at little to no penalty in terms of solution speed.

Cuts

Cuts are constraints added to a model to restrict (cut away) noninteger solutions that would otherwise be solutions of the continuous relaxation. The addition of cuts usually reduces the number of branches needed to solve a MIP.

In the following descriptions of cuts, the term *subproblem* includes the root node (that is, the root relaxation). Cuts are most frequently seen at the root node, but they may be added by ILOG CPLEX at other nodes as conditions warrant.

ILOG CPLEX generates its cuts in such a way that they are valid for all subproblems, even when they are discovered during analysis of a particular subproblem. If the solution to a subproblem violates one of the subsequent cuts, ILOG CPLEX may add a constraint to reflect this condition.

Clique Cuts

A *clique* is a relationship among a group of binary variables such that at most one variable in the group can be positive in any integer feasible solution. Before optimization starts, ILOG CPLEX constructs a graph representing these relationships and finds maximal cliques in the graph.

Cover Cuts

If a constraint takes the form of a *knapsack constraint* (that is, a sum of binary variables with nonnegative coefficients less than or equal to a nonnegative right-hand side), then there is a minimal cover associated with the constraint. A *minimal cover* is a subset of the variables of the inequality such that if all the subset variables were set to one, the knapsack constraint would be violated, but if any one subset variable were excluded, the constraint would be satisfied. ILOG CPLEX can generate a constraint corresponding to this condition, and this cut is called a *cover cut*.

Disjunctive Cuts

A MIP problem can be divided into two subproblems with disjunctive feasible regions of their LP relaxations by branching on an integer variable. *Disjunctive cuts* are inequalities valid for the feasible regions of LP relaxations of the subproblems, but not valid for the feasible region of LP relaxation of the MIP problem.

Flow Cover Cuts

Flow covers are generated from constraints that contain continuous variables, where the continuous variables have variable upper bounds that are zero or positive depending on the setting of associated binary variables. The idea of a flow cover comes from considering the

constraint containing the continuous variables as defining a single node in a network where the continuous variables are in-flows and out-flows. The flows will be on or off depending on the settings of the associated binary variables for the variable upper bounds. The flows and the demand at the single node imply a knapsack constraint. That knapsack constraint is then used to generate a cover cut on the flows (that is, on the continuous variables and their variable upper bounds).

Flow Path Cuts

Flow path cuts are generated by considering a set of constraints containing the continuous variables that define a path structure in a network, where the constraints are nodes and the continuous variables are in-flows and out-flows. The flows will be on or off depending on the settings of the associated binary variables.

Gomory Fractional Cuts

Gomory fractional cuts are generated by applying integer rounding on a pivot row in the optimal LP tableau for a (basic) integer variable with a fractional solution value.

Generalized Upper Bound (GUB) Cover Cuts

A *GUB constraint* for a set of binary variables is a sum of variables less than or equal to one. If the variables in a GUB constraint are also members of a knapsack constraint, then the minimal cover can be selected with the additional consideration that at most one of the members of the GUB constraint can be one in a solution. This additional restriction makes the *GUB cover cuts* stronger (that is, more restrictive) than ordinary cover cuts.

Implied Bound Cuts

In some models, binary variables imply bounds on continuous variables. ILOG CPLEX generates potential cuts to reflect these relationships.

Mixed Integer Rounding (MIR) Cuts

MIR cuts are generated by applying integer rounding on the coefficients of integer variables and the right-hand side of a constraint.

Adding Cuts and Re-Optimizing

Each time ILOG CPLEX adds a cut, the subproblem is re-optimized. ILOG CPLEX repeats the process of adding cuts at a node until it finds no further effective cuts. It then selects the branching variable for the subproblem.

Parameters Affecting Cuts

Parameters control the way each class of cuts is used. Those parameters are listed in Table 13.8.

Table 13.8 *Parameters for Controlling Cuts*

Cut Type	Interactive Command	Concert Technology Parameter	Callable Library Parameter
Clique	`set mip cuts cliques`	`Cliques`	`CPX_PARAM_CLIQUES`
Cover	`set mip cuts covers`	`Covers`	`CPX_PARAM_COVERS`
Disjunctive	`set mip cuts disjunctive`	`DisjCuts`	`CPX_PARAM_DISJCUTS`
Flow Cover	`set mip cuts flowcuts`	`FlowCovers`	`CPX_PARAM_FLOWCOVERS`
Flow Path	`set mip cuts pathcut`	`FlowPaths`	`CPX_PARAM_FLOWPATHS`
Gomory	`set mip cuts gomory`	`FracCuts`	`CPX_PARAM_FRACCUTS`
GUB Cover	`set mip cuts gubcovers`	`GUBCovers`	`CPX_PARAM_GUBCOVERS`
Implied Bound	`set mip cuts implied`	`ImplBd`	`CPX_PARAM_IMPLBD`
Mixed Integer Rounding (MIR)	`set mip cuts mircut`	`MIRCuts`	`CPX_PARAM_MIRCUTS`

The default value of each of those parameters is `0` (zero). By default, ILOG CPLEX automatically determines how often (if at all) it should try to generate that class of cut. A setting of `-1` indicates that no cuts of the class should be generated; a setting of `1` indicates that cuts of the class should be generated moderately; and a setting of `2` indicates that cuts of the class should be generated aggressively. For disjunctive cuts, a setting of 3 is permitted, which indicates that disjunctive cuts should be generated very aggressively.

In the Interactive Optimizer, the command `set mip cuts all i` applies the value `i` to all classes of cut parameters. That is, you can set them all at once.

The `CutsFactor` parameter controls the number of cuts ILOG CPLEX adds to the model. The problem can grow to `CutsFactor` times the original number of rows in the model (or in the presolved model, if the presolver is active). Thus, a `CutsFactor` of 1.0 would mean that no cuts will be generated, which may be a more convenient way of turning off all cuts than setting them individually. The default `CutsFactor` value of 4.0 works well in most cases, as it allows a generous number of cuts while in rare instances it also serves to limit unchecked growth in the problem size.

The `AggCutLim` parameter controls the number of constraints allowed to be aggregated for generating MIR and flow cover cuts.

The FracPass parameter controls the number of passes for generating Gomory fractional cuts. This parameter will not have any effect if the parameter for `set mip cuts gomory` has a non-default value.

The FracCand parameter controls the number of variable candidates to be considered for generating Gomory fractional cuts.

Heuristics

In ILOG CPLEX, a *heuristic* is a procedure that tries to produce good or approximate solutions to a problem quickly but which lacks theoretical guarantees. In the context of solving a MIP, a heuristic is a method that may produce one or more solutions, satisfying all constraints and all integrality conditions, but lacks an indication of whether it has found the best solution possible.

IILOG CPLEX provides two families of heuristics to find integer solutions at nodes (including the root node) during the branch & cut procedure:

- *Node Heuristic* on page 321
- *Relaxation Induced Neighborhood Search (RINS) Heuristic* on page 321

Being integrated into branch & cut, these heuristic solutions gain the same advantages toward a proof of optimality as any solution produced by branching, and in many instances, they can speed the final proof of optimality, or they can provide a suboptimal but high-quality solution in a shorter amount of time than by branching alone. With default parameter settings, ILOG CPLEX automatically invokes the heuristics when they seem likely to be beneficial.

Node Heuristic

The node heuristic employs techniques to try to construct a feasible solution from the current (fractional) branch & cut node. This feature is controlled by the parameter HeurFreq. At its default value of 0, ILOG CPLEX dynamically determines the frequency with which the heuristic is invoked. The setting -1 turns the feature off. A positive value specifies the frequency (in node count) with which the heuristic will be called. For example, if the HeurFreq parameter is set to 20, then the node heuristic will be applied at node 0, node 20, node 40, and so on.

Relaxation Induced Neighborhood Search (RINS) Heuristic

Relaxation induced neighborhood search (RINS) is a heuristic that explores a neighborhood of the current incumbent solution to try to find a new, improved incumbent. It formulates the neighborhood exploration as another MIP (known as the subMIP), and truncates the subMIP optimization by limiting the number of nodes explored in the search tree.

Two parameters apply specifically to RINS: RINSHeur and SubMIPNodeLim.

`RINSHeur` controls how often RINS is invoked, in a manner analogous to the way that `HeurFreq` works. A setting of 100, for example, means that RINS is invoked every 100th node in the tree, while -1 turns it off. The default setting is 0 (zero), which means that ILOG CPLEX decides when to apply it; with this automatic setting, RINS is applied very much less frequently than the node heuristic is applied because RINS typically consumes more time. Also, with the default setting, RINS is turned entirely off if the node heuristic has been turned off via a `HeurFreq` setting of -1; with any other `RINSHeur` setting than 0 (zero), the `HeurFreq` setting does not affect RINS frequency.

`SubMIPNodeLim` restricts the number of nodes searched in the subMIP during application of the relaxation induced neighborhood search (RINS) heuristic. Its default is 500 is satisfactory in most situations, but you can set this parameter to any positive integer if you need to tune performance for your problem.

Probing

While most of the suggestions in this section are oriented toward overcoming specific obstacles, the *probing* feature can help in many different ways on difficult models. Probing is a technique that looks at the logical implications of fixing each binary variable to 0 or 1. It is performed after preprocessing and before the solution of the root relaxation. Probing can be expensive, so this parameter should be used selectively. On models that are in some sense easy, the extra time spent probing may not reduce the overall time enough to be worthwhile. On difficult models, probing may incur very large runtime costs at the beginning and yet pay off with shorter overall runtime. When you are tuning performance, it is usually because the model is difficult, and then probing is worth trying.

At the `Probe` parameter's default setting of `0`, ILOG CPLEX will automatically determine the appropriate level of probing. Setting the `Probe` parameter to `1`, `2`, or `3`, results in increasing levels of probing to be performed beyond the default level of probing. A setting of `-1` results in no probing being performed.

To activate probing::

- In the Interactive Optimizer, use the command `set mip strategy probe i`.
- In Concert Technology, set the integer parameter `Probe`.
- In the Callable Library, set the integer parameter `CPX_PARAM_PROBE`.

Preprocessing: Presolver and Aggregator

When you invoke the MIP optimizer, whether through the Interactive Optimizer command `mipopt`, through a call to the Concert Technology `IloCplex` method `solve`, or through the Callable Library routine `CPXmipopt`—ILOG CPLEX by default automatically preprocesses your problem. Table 13.9 summarizes the preprocessing parameters. In preprocessing, ILOG CPLEX applies its presolver and aggregator one or more times to reduce the size of

the integer program in order to strengthen the initial linear relaxation and to decrease the overall size of the mixed integer program.

Table 13.9 *Parameters for Controlling MIP Preprocessing*

Interactive Command	Concert Technology Parameter	Callable Library Parameter	Comment
`set preprocessing aggregator`	`AggInd`	`CPX_PARAM_AGGIND`	on by default
`set preprocessing presolve`	`PreInd`	`CPX_PARAM_PREIND`	on by default
`set preprocessing boundstrength`	`BndStrenInd`	`CPX_PARAM_BNDSTRENIND`	presolve must be on
`set preprocessing coeffreduce`	`CoeRedInd`	`CPX_PARAM_COEREDIND`	presolve must be on
`set preprocessing relax`	`RelaxPreInd`	`CPX_PARAM_RELAXPREIND`	applies to relaxation
`set preprocessing reduce`	`Reduce`	`CPX_PARAM_REDUCE`	all on by default
`set preprocessing numpass`	`PrePass`	`CPX_PARAM_PREPASS`	automatic by default

The parameters `Reduce` and `Numpass` have the same meanings for LP, QCP, QP and MIP. *Preprocessing* on page 188 explains the meanings and adjustments of all these parameters.

While preprocessing, ILOG CPLEX also attempts to strengthen bounds on variables. This bound strengthening may take a long time. In such cases, you may want to turn off bound strengthening.

ILOG CPLEX also attempts to reduce coefficients during preprocessing. Coefficient reduction usually strengthens the continuous relaxation and reduces the number of nodes in the branch & cut tree, but not always. Sometimes, it increases the amount of time needed to solve the linear relaxations at each node—enough time to offset the benefit of fewer nodes. Two levels of coefficient reduction are available, so it is worthwhile to experiment with these preprocessing options to see whether they are beneficial to your problem.

In addition, you may also set the relaxation parameter to tell ILOG CPLEX to apply preprocessing to the initial relaxation of the problem. Sometimes this preprocessing will result in additional, beneficial presolve transformations in the continuous LP, QP, or QCP relaxation, transformations that are not possible in the original MIP model. In general, MIP preprocessing reductions and continuous LP, QP, or QCP reductions are not the same, as some MIP reductions are not applicable to the continuous case, and some continuous reductions do not apply in the MIP case. Thus, the `RelaxPreInd` parameter allows you to get the benefit of both styles of preprocessing, when the model is large enough that the root solution is itself a time consuming step.

ILOG CPLEX preprocesses a MIP by default. However, if you use a basis to start LP optimization of the root relaxation, ILOG CPLEX will proceed with that starting basis

without preprocessing it. Frequently the strategic benefits of MIP presolve outweigh the tactical advantage of using a starting basis for the root node, so use caution when considering the advantages of a starting basis.

Starting from a Solution

You can provide a known solution (for example, from a MIP problem previously solved or from your knowledge of the problem) to serve as the first integer solution. In such a start, you must specify values for all integer variables, for all semi-continuous variables, and for all members of special ordered sets. Optionally, you may also specify values for continuous variables. ILOG CPLEX evaluates that start solution for integrality and feasibility. If it is integer-feasible, it will become an integer solution of the current problem.

Once a MIP Start has been established for your model, its use is controlled by the `MIPStart` parameter. At its default setting of `0`, the MIP Start values are not used, but you can direct ILOG CPLEX to evaluate the MIP Start values by setting the `MIPStart` parameter to `1`.

You can establish MIP starting values by using the method `setVectors` in a Concert program, or by using `CPXcopymipstart` in a Callable Library program.

For Interactive Optimizer use, or as an alternative approach in a Callable Library program, you can establish MIP starting values from a file. `MST` format (described briefly in the reference manual *ILOG CPLEX File Formats*) is used for this purpose. Use the routine `CPXreadcopymipstart` in the Callable Library, the method `readMIPStart` in Concert Technology, or the `read` command in the Interactive Optimizer, for this purpose.

At the end of a MIP optimization call, when a feasible (not necessarily optimal) solution is still in memory, you can create an `MST` file from the Callable Library with the routine `CPXmstwrite`, from Concert Technology with the method `writeMIPStart`, or from the Interactive Optimizer with the `write` command, for later use as starting values to another MIP problem. Care should be taken to make sure that the naming convention for the variables is consistent between models, when this approach is used.

Priority Orders

In branch & cut, ILOG CPLEX makes decisions about which variable to branch on at a node. You can control the order in which ILOG CPLEX branches on variables by issuing a *priority order*. A priority order assigns a branching priority to some or all of the integer variables in a model. ILOG CPLEX performs branches on variables with a higher assigned priority number before variables with a lower priority; variables not assigned an explicit priority value by the user are treated as having a priority value of `0`. Note that ILOG CPLEX will branch only on variables that take a fractional solution value at a given node. Thus a variable with a high priority number might still not be branched upon until late in the tree, if at all, and indeed if the presolve or the aggregator feature of the ILOG CPLEX Preprocessor removes a given variable then branching on that variable would never occur regardless of a high priority order assigned to it by the user.

You can specify priority for any variable, though the priority is used only if the variable is a general integer variable, a binary integer variable, a semi-continuous variable, a semi-integer variable, or a member of a special ordered set. To specify priority, use one of the following routines or methods:

- From the Callable Library, use `CPXcopyorder` to copy a priority order and apply it, or `CPXreadcopyorder` to read the copy order from a file in ORD format. That format is documented in the reference manual *ILOG CPLEX File Formats*.
- From Concert Technology, use the method `setPriority` to set the priority of a given variable or `setPriorities` to set priorities for an array of variables. Use the method `readOrder` to read priorities from a file in ORD format and apply them.

ILOG CPLEX can generate a priority order automatically, based on problem-data characteristics. This facility can be activated by setting the `MIPOrdType` parameter to one of the values in Table 13.10.

Table 13.10 *Parameters for Branching Priority Order*

Parameter	Branching Priority Order
`0`	no automatic priority order will be generated (default)
`1`	decreasing cost coefficients among the variables
`2`	increasing bound range among the variables
`3`	increasing cost per matrix coefficient count among the variables

If you explicitly read a file of priority orders, its settings will override any generic priority order you may have set by interactive commands.

The parameter `MIPOrdInd`, when set to 0 (zero), allows you to direct ILOG CPLEX to ignore a priority order that was previously read from a file. The default setting for this parameter means that a priority order will be used, if one has been read in.

Problems that use integer variables to represent different types of decisions should assign higher priority to those that must be decided first. For example, if some variables in a model activate processes, and others use those activated processes, then the first group of variables should be assigned higher priority than the second group. In that way, you can use priority to achieve better solutions.

Setting priority based on the magnitude of objective coefficients is also sometimes helpful.

Termination

ILOG CPLEX terminates MIP optimization under a variety of circumstances. First, ILOG CPLEX declares integer optimality and terminates when it finds an integer solution and all nodes have been processed. Optimality in this case is relative to whatever tolerances

and optimality criteria you have set. For example, ILOG CPLEX considers the cutoff value and the objective difference parameter in this context.

In addition, ILOG CPLEX terminates optimization when it reaches a limit that you have set. You can set limits on time, number of nodes, size of tree memory, and number of integer solutions. Table 13.11 summarizes those parameters and their purpose.

Table 13.11 *Parameters to Limit MIP Optimization*

To set a limit on	Use this parameter		
	Concert Technology	**Callable Library**	**Interactive Optimizer**
elapsed time	`TiLim`	`CPX_PARAM_TILIM`	`timelimit`
number of nodes	`NodeLim`	`CPX_PARAM_NODELIM`	`mip limits nodes`
size of tree	`TreLim`	`CPX_PARAM_TRELIM`	`mip limits treememory`
size of node file	`WorkMem`	`CPX_PARAM_WORKMEM`	`workmem`
number of integer solutions	`IntSolLim`	`CPX_PARAM_INTSOLLIM`	`mip limits solutions`

ILOG CPLEX also terminates when an error occurs, such as when ILOG CPLEX runs out of memory or when a subproblem cannot be solved. If an error is due to failure to solve a subproblem, an additional line appears in the node log file to indicate the reason for that failure.

Writing a Tree File

When ILOG CPLEX terminates a MIP optimization before it achieves optimality (for example, because it has reached a limit you set), it still has significant information about the current branch & cut tree. You can save this information by writing it to a file of type TRE (a binary, proprietary ILOG CPLEX format). Later, you can then read the saved TRE file and restart the optimization from where ILOG CPLEX left off.

To save a MIP in a TRE file:

- In the Interactive Optimizer, use the command `write filename.tre` or `write filename tre`.
- From the Callable Library, use the routine `CPXtreewrite`.
- From Concert Technology, use the method `writeTree`.

Note that resuming a branch & cut optimization by reading a tree file back in may not result in precisely the same solution path being taken as though the run had not been interrupted, due simply to the dynamic nature of the branch & cut algorithm.

A TRE file may be quite large (corresponding to the current size of an active tree) so it may consume considerable disk space.

If you modify the model of a MIP *after* you create its TRE file, then the TRE file will be of no use to you. ILOG CPLEX will accept the old TRE file if the basic dimensions of the problem have not changed, but the results it produces from it will likely be invalid for the modified model.

Using the MIP Solution

After you have solved a MIP, you will usually want to make use of the solution in some way. If you are interested only in the values of the variables at the optimum, then you can perform some simple steps to get that information:

- In Concert Technology, the method `getValues` accesses this information.
- In the Callable Library, use the routine `CPXgetmipx`.

After your program has placed the solution values into arrays in this way, it can print the values to the screen, write the values to a file, perform computations using the values, and so forth.

In the Interactive Optimizer, you can print the nonzero solution values to the screen with the command `display solution variables`. A copy of this information goes to the log file, named `cplex.log` by default. Thus one way to print your solution to a file is to temporarily rename the log file. For example, the following series of commands in the Interactive Optimizer will place the solution values of all variables whose values are not zero into a file named `solution.asc`:

```
set logfile solution.asc
display solution variables
set logfile cplex.log
```

For any of the MIP problem types, the following additional solution information is available after optimization:

- objective function value for the best integer solution, if one exists;
- best bound, that is, best objective function value among remaining subproblems;
- solution quality;
- primal values for the best integer solution, if one has been found;
- slack values for best integer solution, if one has been found.

If you request other solution information than these items for a MIP, an error status will be issued. For example, in the Interactive Optimizer, you would get the following message:

```
Not available for mixed integer problems-
use CHANGE PROBLEM to change the problem type
```

Such post-solution information does not have the same meaning in a mixed integer program (MIP) as in a linear program (LP) because of the special nature of the integer variables in the MIP. The reduced costs, dual values, and sensitivity ranges give you information about the effect of making small changes in problem data so long as feasibility is maintained. Integer variables, however, lose feasibility if a small change is made in their value, so this post-solution information cannot be used to evaluate changes in problem data in the usual way of continuous models.

Integer variables often represent major structural decisions in a model, and many continuous variables of the model may be related to these major decisions. With that observation in mind, if you take the integer variable solution values as given, then you can obtain useful post-solution information, applying only to the continuous variables, in the usual way. This is the idea behind the so-called "fixed MIP" problem, a version of the MIP problem where all of the discrete variables are placed at values corresponding to the MIP solution, and thus it is a continuous problem though not strictly a relaxation of the MIP.

Interpreting Solution Statistics on page 203 explains how to use the display command in the Interactive Optimizer to see post-solution information from the linear optimizers. In particular, an LP model having a basic solution (that is, a solution obtained by one of the simplex optimizers, or by the barrier optimizer using crossover) can have its solution written to a file in a standard ASCII-text format, referred to as TXT format. If your discrete model is MILP, then it is natural to wish to use the TXT format for your integer solution, too. However, the TXT format contains information about the dual values of a solution, which you have just seen are not directly available for a MIP solution.

What is needed is to construct the fixed MIP problem, solve it, and then use that solution. (Typically, solving the fixed MIP problem is very rapid compared to solving the MIP itself.)

- In the Interactive Optimizer, follow these steps (*after* optimizing your MIP problem!) to create a TXT format file named `solution.txt`:

  ```
  change problem fixed_milp
  optimize
  write solution.txt
  ```

- In the Callable Library, you accomplish these steps by first calling `CPXchgprobtype`, passing the value `CPXPROB_FIXEDMILP` as the value for the `type` argument, and then calling `CPXlpopt` to re-optimize the resulting LP problem, and finally calling `CPXwritesol` to create your TXT file.

- In Concert Technology, there is no explicit problem type, and so to write a TXT format file, you call the method `solveFixed` followed by the method `writeTextSolution`.

If your problem type is MIQP or MIQCP then this discussion of TXT files is **not** applicable, because TXT format does not support the underlying QP and QCP continuous problem types. Indeed, there does not even exist a problem type of `FIXEDMIQCP` because a continous QCP solution does not contain appropriate dual information. For MIQP, the associated `FIXEDMIQP` problem type is useful for sensitivity analysis, as is the case for `FIXEDMILP`.

Using Special Ordered Sets

A special ordered set (SOS) is an additional way to specify integrality conditions in a model. There are various types of SOS:

- SOS Type 1 is a set of variables (whether all integer, all continuous, or mixed integer and continuous) where at most one variable may be nonzero.
- SOS Type 2 is a set or integer or continuous variables where at most two variables may be nonzero. If two variables are nonzero, they must be adjacent in the set.

ILOG CPLEX uses special branching strategies to take advantage of SOSs. The special branching strategies depend upon the order among the variables in the set. The order is specified by assigning weights to each variable. The order of the variables in the model (such as in the MPS or LP format data file, or the column index in a Callable Library application) is ***not*** used in SOS branching. If there is no order relationship among the variables (such that weights cannot be specified or would not be meaningful), SOS branching should not be used. For many classes of problems, these branching strategies can significantly improve performance.

Example: SOS Type 1 for Sizing a Warehouse

To give you a feel for how SOSs can be useful, here's an example of an SOS Type 1 used to choose the size of a warehouse. Assume for this example that a warehouse of *10000*, *20000*, *40000*, or *50000* square feet can be built. Define binary variables for the four sizes, say, x_1, x_2, x_4, and x_5. Connect these variables by a constraint defining another variable to denote available square feet, like this: $z - 10000x_1 - 20000x_2 - 40000x_4 - 50000x_5 = 0$.

Those four variables are members of a special ordered set. Only one size can be chosen for the warehouse; that is, at most one of the *x* variables can be nonzero in the solution. And, there is an order relationship among the *x* variables (namely, the sizes) that can be used as weights. Then the *weights* of the set members are *10000*, *20000*, *40000*, and *50000*.

Assume furthermore that there is a known fractional (that is, noninteger) solution of $x_1 = 0.1$, $x_5 = 0.9$. These values indicate that other parts of the model have imposed the requirement of *46000* square feet since $0.1*10000 + 0.9*50000 = 46000$. In SOS parlance, the *weighted average* of the set is $(0.1*10000 + 0.9*50000)/(0.1 + 0.9) = 46000$.

Split the set before the variable with weight exceeding the weighted average. In this case, split the set like this: x_1, x_2, and x_4 will be in one subset; x_5 in the other.

Now branch. One branch restricts x_1, x_2, x_4 to *0* (zero). This branch results in x_5 being set to *1* (one).

The other branch, where x_5 is set to *0* (zero), results in an infeasible solution, so it is removed from further consideration.

If a warehouse must be built, then the additional constraint is needed that $x_1 + x_2 + x_4 + x_5 = 1$. The implicit constraint for an SOS Type 1 is less than or equal to one. The continuous relaxation may more closely resemble the MIP if that constraint is added.

Declaring SOS Members

ILOG CPLEX offers you several ways to declare an SOS in a problem:

- Use an SOS file, that is one in SOS format, with the file extension `.sos`. SOS files offer you the most powerful and flexible alternative because the SOS file structure allows you to do several tasks at once:
  - provide branching priorities for sets,
  - assign weights to individual set members,
  - define overlapping sets.
- Use SOS declarations within an MPS or LP file (that is, one in MPS format with the file extension `.mps` or in LP format with the file extension `.lp`. If you already have MPS files with SOS information, you may prefer this option. Conventions for declaring SOS information in MPS files are documented in the *ILOG CPLEX Reference Manual*.

Setting Branching Priority for an SOS

An entire SOS can be given a branching priority. There are two alternative ways to give an SOS branching priority, both documented in the *ILOG CPLEX Reference Manual*:

- Use an SOS file to set priorities.
- Use an ORD file to set priorities.

ILOG CPLEX derives the branching priority of a *set* from the branching priorities of its *members*: the entire set is assigned the highest priority among its members.

To specify SOS priorities:

- In Concert Technology, use the `IloCplex` methods `setPriority` and `setPriorities`.
- In the Callable Library, use the routines `CPXcopysos` or `CPXcopyorder`.

Assigning SOS Weights

Members of an SOS should be given unique weights that in turn define the order of the variables in the set. (These unique weights are also called *reference row values*.) The most flexible way for you to assign weights is through an SOS, MPS, or LP file. An alternative is to use MPS format to assign a single reference row containing weights. Such a reference row may be a free row with specific weighting information, or it may be the objective function, or it may be a constraint row. In the Callable Library, the routine `CPXcopysos` lets you specify weights directly in an application.

The SOS example, *Example: SOS Type 1 for Sizing a Warehouse* on page 329, used the coefficients of the warehouse capacity constraint to assign weights.

Using Semi-Continuous Variables

Semi-continuous variables are variables that may take either the value 0 (zero) or a value within a finite range [*a, b*]. Semi-continuous variables can be specified in MPS and LP files. *Stating a MIP Problem* on page 306 tells you how to specify variables as semi-continuous.

In the Callable Library, semi-continuous variables can be entered with type `CPX_SEMICONT` or `CPX_SEMIINT` via the routine `CPXcopyctype`. In that case, the lower bound of 0 (zero) is implied; the semi-continuous lower bound is defined by the corresponding entry in the array of lower bounds; and likewise, the semi-continuous upper bound is defined by the corresponding entry in the array of upper bounds of the problem.

In Concert Technology, semi-continuous variables are instances of the class `IloSemiContVar`. For a Concert Technology example using semi-continuous variables, see also *Rates: Using Semi-Continuous Variables* on page 389 in this manual.

Progress Reports: Interpreting the Node Log

As explained earlier, when ILOG CPLEX optimizes mixed integer programs, it builds a tree with the linear relaxation of the original MIP at the root and subproblems to optimize at the nodes of the tree. ILOG CPLEX reports its progress in optimizing the original problem in a *node log file* as it traverses this tree.

You control how information in the log file is recorded and displayed, through two ILOG CPLEX parameters. The `MIPDisplay` parameter controls the general nature of the output that goes to the node log. Table 13.12 summarizes its possible values and their effects.

Table 13.12 *Settings of the MIP Display Parameter*

Setting	Effect
0	no display
1	display integer feasible solutions
2	display nodes under mip interval control
3	same as 2, but add information on node cuts
4	same as 3, but add LP display for root node
5	same as 3, but add LP display for all nodes

The `MIPInterval` parameter controls how frequently node log lines are printed. Its default is `100` and can be set to any positive integer value. A line is recorded in the node log for every node with an integer solution if the `MIPDisplay` parameter is set to `1` or higher, and also for any node whose number is a multiple of the `MIPInterval` value if the `MIPDisplay` is set to `2` or higher.

Here is an example of a log file from the Interactive Optimizer, where the `MIPInterval` parameter has been set to 10:

```
Tried aggregator 1 time.
No MIP presolve or aggregator reductions.
Presolve time =    0.00 sec.
Root relaxation solution time = 0.00 sec
Objective is integral.

     Nodes                                         Cuts/
  Node Left  Objective  IInf  Best Integer   Best Node  ItCnt Gap

     0   0     4.0000     6                     4.0000     12
*    4   2                0        5.0000       4.0000     17 20.00%
    10   1     cutoff             5.0000       4.0000     31 20.00%

Integer optimal solution:  Objective =   5.0000000000e+000
Solution time =    0.02 sec.    Iterations = 41   Nodes = 13
```

In that example, ILOG CPLEX found the optimal objective function value of 5.0 after exploring 13 nodes and performing 41 (dual simplex) iterations, and ILOG CPLEX found an optimal integer solution at node 4. The MIP interval parameter was set at 10, so every tenth node was logged, in addition to the node where an integer solution was found.

As you can see in that example, ILOG CPLEX logs an asterisk (`*`) in the left-most column for any node where it finds an integer-feasible solution. In the next column, it logs the *node number*. It next logs the number of nodes left to explore.

In the next column, `Objective`, ILOG CPLEX either records the *objective value* at the node or a *reason to fathom* the node. (A node is fathomed if the solution of a subproblem at the node is infeasible; or if the value of objective function at the node is worse than the cutoff value for branch & cut; or if the node supplies an integer solution.) This column is left blank for lines where the first column contains an asterisk (*) indicating a node where ILOG CPLEX found an integer-feasible solution.

In the column labeled `IInf`, ILOG CPLEX records the number of integer-infeasible variables and special ordered sets. If no solution has been found, the next column is left blank; otherwise, it records the best integer solution found so far.

The column labeled `Cuts/Best Node` records the best objective function value achievable. If the word `Cuts` appears in this column, it means various cuts were generated; if a particular name of a cut appears, then only that kind of cut was generated.

The column labeled `ItCnt` records the cumulative iteration count of the algorithm solving the subproblems. Until a solution has been found, the column labeled `Gap` is blank. If a solution has been found, the relative gap value is printed when it is less than `999.99`; otherwise, hyphens are printed. The gap is computed as `abs(best integer - best node)/(1e-10 + abs(best integer))`. Consequently, the printed gap value may not always move smoothly. In particular, there may be sharp improvements whenever a new best integer solution is found.

ILOG CPLEX also logs its addition of cuts to a model. Here is an example of a node log file from a problem where ILOG CPLEX added several cover cuts.

```
MIP Presolve eliminated 0 rows and 1 columns.
MIP Presolve modified 12 coefficients.
Reduced MIP has 15 rows, 32 columns, and 97 nonzeros.
Presolve time =    0.00 sec.

    Nodes                                         Cuts/
 Node  Left   Objective  IInf  Best Integer  Best Node  ItCnt Gap

    0     0   2819.3574     7                2819.3574     35
                2881.8340     8                Covers: 4     44
                2881.8340    12                Covers: 3     48
*   7     6                  0     3089.0000  2904.0815     62 5.99%

Cover cuts applied:  30

Integer optimal solution:  Objective =   3.0890000000e+003
Solution time =    0.10 sec.    Iterations = 192   Nodes = 44
```

ILOG CPLEX also logs the number of clique inequalities in the clique table at the beginning of optimization. Cuts generated at intermediate nodes are not logged individually unless they happen to be generated at a node logged for other reasons. ILOG CPLEX logs the number of applied cuts of all classes at the end.

ILOG CPLEX also indicates, in the node log file, each instance of a successful application of the node heuristic. The following example shows a node log file for a problem where the heuristic found a solution at node 0. The + denotes a node generated by the heuristic.

```
 Nodes                                          Cuts/
  Node  Left   Objective  IInf  Best Integer  Best Node  ItCnt Gap
    0     0      403.8465  640                 403.8465   4037
                 405.2839  609               Cliques: 10   5208
                 405.2891  612               Cliques:  2   5288
Heuristic: feasible at 437.000, still looking
Heuristic: feasible at 437.000, still looking
Heuristic complete
*   0+    0                  0      436.0000   405.2891   5288 7.04%
```

Periodically, if the MIP display parameter is 2 or greater, ILOG CPLEX records the cumulative time spent since the beginning of the current MIP optimization and the amount of memory used by branch & cut. (Periodically means that time and memory information appear either every 20 nodes or ten times the MIP display parameter, whichever is greater.) The following example shows you one line from a node log file indicating elapsed time and memory use.

```
Elapsed b&c time = 120.01 sec. (tree size =  0.09 MB)
```

The Interactive Optimizer prints an additional summary line in the log if optimization stops before it is complete. This summary line shows the best MIP bound, that is, the best objective value among all the remaining node subproblems. The following example shows you lines from a node log file where an integer solution has not yet been found, and the best remaining objective value is 2973.9912281.

```
Node limit, no integer solution.
Current MIP best bound =    2.9739912281e+03 (gap is infinite)
Solution time =    0.01 sec.  Iterations = 68  Nodes = 7 (7)
```

Stating a MIP Problem on page 306 presents a typical MIP problem. Here is the node log file for that problem with the default setting of the MIP display parameter:

```
Tried aggregator 1 time.
Aggregator did 1 substitutions.
Reduced MIP has 2 rows, 3 columns, and 6 nonzeros.
Presolve time =    0.00 sec.
Clique table:0 GUB, 0 GUBEQ, 0 two-covers, 0 probed
ImplBd table: 0 bounds
Root relaxation solution time =    0.00 sec.

      Nodes                                         Cuts/
 Node  Left  Objective  IInf  Best Integer  Best Node  ItCnt Gap

   0     0    125.2083     1                125.2083      3
*                         0      122.5000   Cuts:  2      4

Mixed integer rounding cuts applied: 1

Integer optimal solution:  Objective =   1.2250000000e+002
Solution time =    0.02 sec.    Iterations = 4   Nodes = 0
```

These additional items appear only in the node log file (not on screen):

- `Variable` records the name of the variable where ILOG CPLEX branched to create this node. If the branch was due to a special ordered set, the name listed here will be the right-most variable in the left subset.
- `B` indicates the branching direction:
  - `D` means the variables was restricted to a lower value;
  - `U` means the variable was restricted to a higher value;
  - `L` means the left subset of the special ordered set was restricted to 0 (zero);
  - `R` means the right subset of the special ordered set was restricted to 0 (zero).
  - `A` means that constraints were added or more than one variable was restricted.
- `Parent` indicates the node number of the parent.
- `Depth` indicates the depth of this node in the branch & cut tree.

Troubleshooting MIP Performance Problems

Even the most sophisticated methods currently available to solve pure integer and mixed integer programming problems require noticeably more computation than the methods for similarly sized continuous problems. Many relatively small integer programming models still take enormous amounts of computing time to solve. Indeed, some such models have

never yet been solved. In the face of these practical obstacles to a solution, proper formulation of the model is crucial to successful solution of pure integer or mixed integer programs.

For help in formulating a model of your own integer or mixed integer problem, you may want to consult H.P. Williams's textbook about practical model building (referenced in *Further Reading* on page 36 in the preface of this manual).

Also you may want to develop a better understanding of the branch & cut algorithm. For that purpose, Williams's book offers a good introduction, and Nemhauser and Wolsey's book (also referenced in *Further Reading* on page 36 in the preface of this manual) goes into greater depth about branch & cut as well as other techniques implemented in the ILOG CPLEX MIP Optimizer.

Performance Features of the Mixed Integer Optimizer on page 313 in this chapter has already discussed several specific features that are important for performance tuning of difficult models. Here are more specific performance symptoms and the remedies that can be tried.

- *Too Much Time at Node 0* on page 336
- *Trouble Finding More than One Feasible Solution* on page 337
- *Large Number of Unhelpful Cuts* on page 337
- *Lack of Movement in the Best Node* on page 337
- *Time Wasted on Overly Tight Optimality Criteria* on page 338
- *Running Out of Memory* on page 340
- *Difficulty Solving Subproblems* on page 343
- *Subproblem Optimization* on page 344

Too Much Time at Node 0

If you observe that a very long time passes before the branch & cut algorithm begins processing nodes, it may be that the root relaxation problem itself is taking a long time. The standard screen display will print a line telling "`Root relaxation solution time =`" once this root solve is complete, and a large solution time would be an indicator of an opportunity for tuning. If you set the `MIPDisplay` parameter to `4`, you may get a further indication of the difficulties this root solve has run into. Tuning techniques found in Chapter 8, *Solving LPs: Simplex Optimizers*, Chapter 11, *Solving Problems with a Quadratic Objective (QP)*, and Chapter 12, *Solving Problems with Quadratic Constraints (QCP)* are applicable to tuning the root solve of a MIP model, too. In particular, it is worth considering setting the `RootAlg` parameter to a nondefault setting, such as the Barrier optimizer, to see if a simple change in algorithm will speed up this step sufficiently.

For some problems, ILOG CPLEX will spend a significant amount of time performing computation at node 0, apart from solving the continuous LP, QP, or QCP relaxation. While this investment of time normally saves in the overall branch & cut, it does not always do so. Time spent at node 0 can be reduced by two parameters.

First, you can try turning off the node heuristic by setting the parameter `HeurFreq` to `-1`. Second, try a less expensive variable selection strategy by setting the parameter `VarSel` to `4`, pseudo reduced costs.

It is worth noting that setting the `MIPEmphasis` parameter to `1`, resulting in an emphasis on feasibility instead of optimality, often also speeds up the processing of the root node. If your purposes are compatible with this emphasis, consider using it.

Trouble Finding More than One Feasible Solution

For some models, ILOG CPLEX finds an integer feasible solution early in the process and then does not find a better one for quite a while. One possibility, of course, is that the first feasible solution is optimal. In that case, there are no better solutions.

One possible approach to finding more feasible solutions is to increase the frequency of the node heuristic, by setting the `HeurFreq` parameter to a value like `10` or `5` or even `1`. This heuristic can be expensive, so exercise caution when setting this parameter to values less than `10`.

Another approach to finding more feasible solutions is to try a node selection strategy alternative. Setting the `NodeSel` parameter to `2` invokes a best-estimate search, which sometimes does a better job of locating good quality feasible solutions than the default node selection strategy.

Large Number of Unhelpful Cuts

While the cuts added by ILOG CPLEX reduce runtime for most problems, on occasion they can have the opposite effect. If you notice, for example, that ILOG CPLEX adds a large number of cuts at the root, but the objective value does not change significantly, then you may want to experiment with turning off cuts.

- In the Interactive Optimizer, you can turn cuts off selectively (`set mip cuts covers -1`) or all at once (`set mip cuts all -1`).
- In the Component Libraries, you can set the parameters that control classes of cuts. The parameters are listed in Table 13.8 on page 320.

Lack of Movement in the Best Node

For some models, the `Best Node` value in the node log changes very slowly or not at all. Runtimes for such models can sometimes be reduced by the variable selection strategy

known as *strong branching*. Strong branching explores a set of candidate branching-variables in-depth, performing a limited number of simplex iterations to estimate the effect of branching up or down on each.

Important: *Strong branching consumes significantly more computation time per node than the default variable selection strategy.*

To activate strong branching, set the `VarSel` parameter to a value of `3`.

On rare occasions, it can be helpful to modify strong branching limits. If you modify the limit on the size of the candidate list, then strong branching will explore a larger (or smaller) set of candidates. If you modify the limit on strong branching iteration, then strong branching will perform more (or fewer) simplex iterations per candidate.

These limits are controlled by the parameters `StrongCandLim` and `StrongItLim`, respectively.

Other parameters to consider trying, in the case of slow movement of the Best Node value, are non-default levels for `Probe` (try the aggressive setting of `3` first, and then reduce it if the probing step itself takes excessive time for your purposes), and `MIPEmphasis` set to a value of `3`.

Time Wasted on Overly Tight Optimality Criteria

Sometimes ILOG CPLEX finds a good integer solution early, but must examine many additional nodes to prove the solution is optimal. You can speed up the process in such a case if you are willing to change the optimality tolerance. ILOG CPLEX supports two kinds of tolerance:

- *Relative optimality tolerance* guarantees that a solution lies within a certain percentage of the optimal solution.
- *Absolute optimality tolerance* guarantees that a solution lies within a certain absolute range of the optimal solution.

The default relative optimality tolerance is 0.0001. At this tolerance, the final integer solution is guaranteed to be within 0.01% of the optimal value. Of course, many formulations of integer or mixed integer programs do not require such tight tolerance, so requiring ILOG CPLEX to seek integer solutions that meet this tolerance in those cases is wasted computation. If you can accept greater optimality tolerance in your model, then you should change the parameter `EpGap`.

If, however, you know that the objective values of your problem are near zero, then you should change the absolute gap because percentages of very small numbers are less useful as optimality tolerance. Change the parameter `EpAGap` in this case.

To speed up the proof of optimality, you can set objective difference parameters, both relative and absolute. Setting these parameters helps when there are many integer solutions

with similar objective values. For example, setting the `ObjDif` parameter to `100.0` makes ILOG CPLEX skip any potential solution with its objective value within 100.0 units of the best integer solution so far. Or, setting the `RelObjDif` to `0.01` would mean that ILOG CPLEX would skip any potential new solution that is not at least 1% better than the incumbent solution. Naturally, since this objective difference setting may make ILOG CPLEX skip an interval where the true integer optimum may be found, the objective difference setting weakens the guarantee of optimality.

Cutoff parameters can also be helpful in restricting the search for optimality. If you know that there are solutions within a certain distance of the initial relaxation of your problem, then you can readily set the upper cutoff parameter for minimization problems and the lower cutoff parameter for maximization problems. Set the parameters `CutUp` and `CutLo`, respectively, to establish a cutoff value.

When you set a MIP cutoff value, ILOG CPLEX searches with the same solution strategy as though it had already found an integer solution, using a node selection strategy that differs from the one it uses before a first solution has been found.

Slightly Infeasible Integer Variables

On some models, the integer solution returned by CPLEX at default settings may contain solution values for the discrete variables that violate integrality by a small amount. The integrality tolerance parameter, `EpInt`, has a default value of 1e-5, which means that any discrete variable that violates integrality by no more than this amount will not be branched upon for resolution. For most model formulations, this situation is satisfactory and avoids branching that may be essentially meaningless and only consumes additional computing time.

However, some formulations combine discrete and continuous variables, for example, involving constraint coefficients over a million in magnitude, where even a small integrality violation can be magnified elsewhere in the model. In such situations, you may attempt to address this variation by tightening the simplex feasibility tolerance, `EpRHS`, from its default value of 1e-6 to as low as 1e-9, its minimum, and also tighten `EpInt` to a similar value, and re-run the MIP optimization from the beginning.

If this adjustment is insufficient to give satisfactory results, you can also try setting `EpInt` all the way to zero, preferably in conjunction with a tightened `EpRHS` setting. This very tight integrality tolerance directs CPLEX to attempt to branch on any integer infeasibility, no matter how small. Numerical roundoff due to floating-point arithmetic on any computer may make it impossible to achieve this tolerance, but in practice, the setting achieves its aim in many models and reduces the integrality violations to near the machine tolerance (around 1e-15) on many others.

Running Out of Memory

A very common difficulty with MIPs is running out of memory. This problem occurs when the branch & cut tree becomes so large that insufficient memory remains to solve a continuous LP, QP, or QCP subproblem. As memory gets tight, you may observe warning messages from ILOG CPLEX as it attempts various operations in spite of limited memory. In such a situation, if ILOG CPLEX does not find a solution shortly, it terminates the process with an error message.

The information about a tree that ILOG CPLEX accumulates in memory can be substantial. In particular, ILOG CPLEX saves a basis for every unexplored node. Furthermore, when ILOG CPLEX uses the best bound or best estimate strategies of node selection, the list of unexplored nodes itself can become very long for large or difficult problems. How large the unexplored node list can be depends on the actual amount of memory available, the size of the problem, and algorithm selected.

A less frequent cause of memory consumption is the generation of cutting planes. Gomory fractional cuts, and, in rare instances, Mixed Integer Rounding cuts, are the ones most likely to be dense and thus use significant memory under default/automatic settings. You can try turning off these cuts, or any of the cuts you see listed as being generated for your model (in the cuts summary at the end of the node log), or simply all cuts, through the use of parameter settings discussed in the section on cuts in this manual; doing this carries the risk that this will make the model harder to solve and only delay the eventual exhaustion of available memory during branching.

Certainly, if you increase the amount of available memory, you extend the problem-solving capability of ILOG CPLEX. Unfortunately, when a problem fails because of insufficient memory, it is difficult to project how much further the process needed to go and how much more memory is needed to solve the problem. For these reasons, the following suggestions aim at avoiding memory failure whenever possible and recovering gracefully otherwise.

Reset the Tree Memory Parameter

To avoid a failure due to running out of memory, set the working memory parameter, `WorkMem`, to a value significantly lower than the available memory on your computer (in megabytes), to instruct ILOG CPLEX to begin compressing the storage of nodes before it consumes all of available memory. See the related topic *Use Node Files for Storage* on page 341, for other choices of what should happen when `WorkMem` is exceeded.

Because the storage of nodes can require a lot of space, it may also be advisable to set a tree limit on the size of the entire tree being stored so that not all of your disk will be filled up with working storage. The call to the MIP optimizer will be stopped once the size of the tree exceeds the value of `TreLim`, the tree limit parameter. Under default settings the limit is infinity ($1e^{+75}$), but you can set it to a lower value (in megabytes)

Write a Tree File and Restart

On some platforms, even when the current tree size is within system limits, memory fragmentation may be so great that performance becomes poor. To overcome that kind of fragmentation, stop optimization, write a tree file (using the TRE format), exit ILOG CPLEX, restart it, read in the model and tree file, and continue optimization then.

Use Node Files for Storage

ILOG CPLEX offers a node file storage feature to store some parts of the branch & cut tree in files. If you use this feature, ILOG CPLEX will be able to explore more nodes within a smaller amount of computer memory. This feature includes several options to reduce the use of physical memory, and it entails a very small increase in runtime. Node file storage offers a much better option than relying on swap space.

This feature is especially helpful when you are using steepest-edge pricing as the subproblem simplex pricing strategy because pricing information itself consumes a great deal of memory.

> *Note: Try node files whenever the MIP optimizer terminates with the condition "out of memory" and the node count is greater than zero. The message in such a situation looks like Code Sample 13.1*


```
Clique cuts applied:  30
CPLEX Error  1001: Out of memory.

Consider using CPLEX node files to reduce memory usage.

Error termination, integer feasible:  Objective =    5.6297000000e+04
Current MIP best bound =    5.5731783224e+04 (gap = 565.217, 1.00%)
Solution time =  220.75 sec.  Iterations = 16707  Nodes = 101 (58)
```

Code Sample 13.1 *Termination Indicating Node Files Would Help*

There are several parameters that control the use of node files. They are:

- `WorkMem`
- `NodeFileInd`
- `TreLim`
- `WorkDir`

When tree storage size exceeds the limit defined by `WorkMem`, what happens next is determined by the setting of `NodeFileInd`. If the latter parameter is set to zero, then optimization proceeds with the tree stored in memory until ILOG CPLEX reaches the tree memory limit (`TreLim`). If the `NodeFileInd` parameter is set to `1` (the default), then a fast compression algorithm is used on the nodes to try to conserve memory, without resorting to writing the node files to disk. If the parameter is set to 2, then node files are written to disk.

If the parameter is set to 3, then nodes are both compressed (as in option 1) and written to disk (as in option 2). Thus, regardless of the setting of `NodeFileInd`, ILOG CPLEX will stop the optimization when the total memory used to store the tree exceeds the tree memory limit Table 13.13 summarizes these different options.

Table 13.13 *Settings for the Node File Storage Parameter*

Setting	Meaning	Comments
`0`	no node files	optimization continues
`1`	node file in memory and compressed	optimization continues (default)
`2`	node file on disk	files created in temporary directory
`3`	node file on disk and compressed	files created in temporary directory

In cases where node files are written to disk, ILOG CPLEX will create a temporary subdirectory under the directory specified by the `WorkDir` parameter. The directory named by this parameter must exist before ILOG CPLEX attempts to create node files. By default, the value of this parameter is ".", which means the current working directory.

ILOG CPLEX creates the temporary directory by means of system calls. If the system environment variable is set (on Windows platforms, the environment variable `TMP`; on UNIX platforms, the environment variable `TMPDIR`), then the system ignores the ILOG CPLEX node-file directory parameter and creates the temporary node-file directory in the location indicated by its system environment variable. Furthermore, if the directory specified in the ILOG CPLEX node-file directory parameter is invalid (for example, if it contains illegal characters, or if the directory does not allow write access), then the system chooses a location according to its own logic.

The temporary directory created for node file storage will have a name prefixed by `cpx`. The files within it will also have names prefixed by `cpx`.

ILOG CPLEX automatically removes the files and their temporary directory when it frees the branch & cut tree:

- in the Interactive Optimizer,
  - at problem modification;
  - at normal termination;
- from Concert Technology,
  - when you call `env.end`
  - when you modify the extracted model
- from the Callable Library,
  - when you call a problem modification routine;

- when you call `CPXfreeprob`.

If a program terminates abnormally, the files are not removed.

ILOG CPLEX uses node file storage most effectively when the amount of working memory is reasonably large so that it does not have to create node files too frequently. A reasonable amount is to use approximately half the memory, but no more than 128 megabytes. Higher values result in only marginally improved efficiency.

Node files could grow very large. Use the `TreMem` parameter to limit the size of the tree so that it does not exceed available disk space, when you choose `NodeFileInd` settings `2` or `3`. It is usually better to let ILOG CPLEX terminate the run gracefully, with whatever current feasible solution has been found, than to trigger an error message or even a program abort.

When ILOG CPLEX uses node-file storage, the sequence of nodes processed may differ from the sequence in which nodes are processed without node-file storage. Nodes in node-file storage are not accessible to user-written callback routines.

Change Algorithms

The best approach to reduce memory use is to modify the solution process. Here are some ways to do so:

- Switch the node selection strategy to best estimate, or more drastically to depth-first, as explained in Table 13.8 on page 320. Depth-first search rarely generates a long, memory-consuming list of unexplored nodes since ILOG CPLEX dives deeply into the tree instead of jumping around. A narrowly focused search, like depth-first, also often results in faster processing times for individual nodes. However, overall solution time is generally much worse than with best-bound node selection because each branch is searched exhaustively to its deepest level before it is fathomed in favor of better branches.
- Another memory-conserving strategy is to use strong branching (`VarSel` parameter value `3`) for variable selection. Strong branching requires substantial computational effort at each node to determine the best branching variable. As a result, it generates fewer nodes and thus makes less overall demand on memory. Often, strong branching is faster as well.

Difficulty Solving Subproblems

There are classes of MIPs that produce very difficult subproblems, for example, if the subproblems are dual degenerate. In such a case, a different optimizer, such as the primal simplex or the barrier optimizer, may be better suited to your problem than the default dual simplex optimizer for subproblems. These alternatives are discussed in *Subproblem Optimization* on page 344. A stronger dual pricing algorithm, such as dual steepest-edge pricing (the parameter `DPriInd` set to `2`), could also be considered.

Overcoming Degeneracy

If the subproblems are dual degenerate, then consider using the primal simplex optimizer for the subproblems. You make this change by setting the `SubAlg` parameter to `1`.

Another effective strategy in overcoming dual degeneracy is to permanently perturb the problem. For subproblems that are dual degenerate, in the Interactive Optimizer, write out the perturbed problem as a DPE file with the command `write filename.dpe` substituting an appropriate file name. (A `.dpe` file is saved as a binary SAV format file.) Then you can read the saved file back in and solve it. The subproblem should then solve more cleanly and quickly.

In the case of DPE files solved by the dual simplex optimizer, any integer solution is also guaranteed to be an integer-feasible solution to the original problem. In most cases, the solution will be optimal or near-optimal as well.

Subproblem Optimization

In some problems, you can improve performance by evaluating how the continous LP, QP, or QCP subproblems are solved at the nodes in the branch & cut tree, and then possibly modifying the choice of algorithm to solve them. You can control which algorithm ILOG CPLEX applies to the initial relaxation of your problem separately from your control of which algorithm ILOG CPLEX applies to other subproblems. The following sections explain those parameters more fully.

RootAlg Parameter

The `RootAlg` algorithm parameter indicates the algorithm for ILOG CPLEX to use on the *initial* subproblem. In a typical MIP, that initial subproblem is usually the linear relaxation of the original MIP. By default, ILOG CPLEX starts the initial subproblem with the dual simplex optimizer. You may have information about your problem that indicates another optimizer could be more efficient. Table 13.14 summarizes the values available for the `RootAlg` parameter.

To set this parameter:

- In the Interactive Optimizer, use the command `set mip strategy startalgorithm` with the value to indicate the optimizer you want.
- In Concert Technology, use the method `IloCplex` method `setParam` with the parameter `RootAlg` and the appropriate algorithm enumeration value.
- In the Callable Library, use the routine `CPXsetintparam` with the parameter `CPX_PARAM_STARTALG`, and the appropriate symbolic constant.

Table 13.14 *Settings of RootAlg and NodeAlg Parameters*

Concert Technology Enumeration	Callable Library Symbolic Constant	Setting	Calls this Optimizer
`Auto`	`CPX_ALG_AUTO`	0	automatic
`Primal`	`CPX_ALG_PRIMAL`	1	primal simplex
`Dual`	`CPX_ALG_DUAL`	2	dual simplex (default)
`Network`	`CPX_ALG_HYBNETOPT`	3	network simplex
`Barrier`	`CPX_ALG_BARRIER`	4	barrier with crossover
`Sifting`	`CPX_ALG_SIFTING`	5	sifting
`Concurrent`	`CPX_ALG_CONCURRENT`	6	concurrent: allowed at root, but not at nodes

NodeAlg Parameter

The `NodeAlg` parameter indicates the algorithm for ILOG CPLEX to use on node relaxations other than the root node. By default, ILOG CPLEX applies the dual simplex optimizer to subproblems, and unlike the `RootAlg` parameter it is extremely unusual for this to not be the most desirable choice, but again, you may have information about your problem that tells you another optimizer could be more efficient. The values and symbolic constants are the same for the `NodeAlg` parameter as for the `RootAlg` parameter in Table 13.14.

To set the `NodeAlg` parameter:

- In Concert Technology, use the `IloCplex` method `setParam` with the parameter `NodeAlg` and the appropriate algorithm enumeration value.
- In the Callable Library, use the routine `CPXsetintparam` with the parameter `CPX_PARAM_SUBALG`, and the appropriate symbolic constant.
- In the Interactive Optimizer, use the command `set mip strategy subalgorithm` with the value to indicate the optimizer you want.

> *Note: Only simplex and barrier optimizers can solve problems of type QP (quadratic term in the objective function).*
>
> *Only the barrier optimizer can solve problems of type QCP (quadratic terms among the constraints).*

Example: Optimizing a Basic MIP Problem

This example illustrates how to optimize a MIP with the ILOG CPLEX Component Libraries.

Complete Program: ilomipex1.cpp

The example derives from `ilolpex8.cpp`. Here are the differences between that linear program and this mixed integer program:

- The problem to solve is slightly different. It appears in *Stating a MIP Problem* on page 306.
- The routine `populatebyrow` added the variables, objective, and constraints to the model created by the method `IloModel model(env)`.

```
// File: examples/src/ilomipex1.cpp
// Version 9.0
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// ilomipex1.cpp - Entering and optimizing a MIP problem

#include <ilcplex/ilocplex.h>

ILOSTLBEGIN

static void
   populatebyrow(IloModel model, IloNumVarArray var, IloRangeArray con);

int
main (void) {
   IloEnv env;
   try {
      IloModel model(env);

      IloNumVarArray var(env);
      IloRangeArray con(env);
      populatebyrow (model, var, con);

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

      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;
```

```
      IloNumArray vals(env);
      cplex.getValues(vals, var);
      env.out() << "Values        = " << vals << endl;
      cplex.getSlacks(vals, con);
      env.out() << "Slacks        = " << vals << endl;

      cplex.exportModel("mipex1.lp");
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();
   return 0;

}  // END main


static void
populatebyrow (IloModel model, IloNumVarArray x, IloRangeArray c)
{
   IloEnv env = model.getEnv();

   x.add(IloNumVar(env, 0.0, 40.0));
   x.add(IloNumVar(env));
   x.add(IloNumVar(env));
   x.add(IloNumVar(env, 2.0, 3.0, ILOINT));
   model.add(IloMaximize(env, x[0] + 2 * x[1] + 3 * x[2] + x[3]));

   c.add( - x[0] +     x[1] + x[2] + 10 * x[3] <= 20);
   c.add(   x[0] - 3 * x[1] + x[2]             <= 30);
   c.add(              x[1]        - 3.5* x[3] == 0);
   model.add(c);

}  // END populatebyrow
```

Complete Program: mipex1.c

The example derives from `lpex8.c`. Here are the differences between that linear program and this mixed integer program:

- The problem to solve is slightly different. It appears in *Stating a MIP Problem* on page 306.
- The routine `setproblemdata` has a parameter, `ctype`, to set the types of the variables to indicate which ones must assume integer values. The routine `CPXcopyctype` associates this data with the problem that `CPXcreateprob` creates.

- The example calls `CPXmipopt` to optimize the problem, rather than `CPXlpopt`.
- The example calls the routines `CPXgetstat`, `CPXgetmipobjval`, `CPXgetmipx`, and `CPXgetmipslack` (instead of `CPXsolution`) to get a solution.

  You do not get dual variables this way. If you want dual variables, you must do the following:

  - Use `CPXchgprobtype` to change the problem type to `CPXPROB_FIXEDMILP`.
  - Then call `CPXprimopt` to optimize that problem.
  - Then use `CPXsolution` to get a solution to the *fixed* problem.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/mipex1.c                                             */
/*  Version 9.0                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2003 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* mipex1.c - Entering and optimizing a MIP problem */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the include of cplex.h. */

#include <ilcplex/cplex.h>

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

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

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

static int
   setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                   int *objsen_p, double **obj_p, double **rhs_p,
                   char **sense_p, int **matbeg_p, int **matcnt_p,
                   int **matind_p, double **matval_p,
                   double **lb_p, double **ub_p, char **ctype_p);

static void
   free_and_null (char **ptr);


/* The problem we are optimizing will have 3 rows, 4 columns
   and 9 nonzeros.  */

#define NUMROWS    3
```

```
#define NUMCOLS    4
#define NUMNZ      9


int
main (void)
{
/* Declare pointers for the variables and arrays that will contain
   the data which define the LP problem.  The setproblemdata() routine
   allocates space for the problem data.  */

   char     *probname = NULL;
   int      numcols;
   int      numrows;
   int      objsen;
   double   *obj = NULL;
   double   *rhs = NULL;
   char     *sense = NULL;
   int      *matbeg = NULL;
   int      *matcnt = NULL;
   int      *matind = NULL;
   double   *matval = NULL;
   double   *lb = NULL;
   double   *ub = NULL;
   char     *ctype = NULL;

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

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


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

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

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

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

/* Turn on output to the screen */

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

/* Fill in the data for the problem.  */

status = setproblemdata (&probname, &numcols, &numrows, &objsen, &obj,
                         &rhs, &sense, &matbeg, &matcnt, &matind, &matval,
                         &lb, &ub, &ctype);
if ( status ) {
   fprintf (stderr, "Failed to build problem data arrays.\n");
   goto TERMINATE;
}

/* Create the problem. */

lp = CPXcreateprob (env, &status, probname);

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

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

/* Now copy the problem data into the lp */

status = CPXcopylp (env, lp, numcols, numrows, objsen, obj, rhs,
                    sense, matbeg, matcnt, matind, matval,
                    lb, ub, NULL);

if ( status ) {
   fprintf (stderr, "Failed to copy problem data.\n");
   goto TERMINATE;
```

```
}

/* Now copy the ctype array */

status = CPXcopyctype (env, lp, ctype);
if ( status ) {
   fprintf (stderr, "Failed to copy ctype\n");
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */

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

solstat = CPXgetstat (env, lp);

/* Write the output to the screen. */

printf ("\nSolution status = %d\n", solstat);

status = CPXgetmipobjval (env, lp, &objval);
if ( status ) {
   fprintf (stderr,"No MIP objective value available.  Exiting...\n");
   goto TERMINATE;
}

printf ("Solution value  = %f\n\n", objval);

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

cur_numrows = CPXgetnumrows (env, lp);
cur_numcols = CPXgetnumcols (env, lp);

status = CPXgetmipx (env, lp, x, 0, cur_numcols-1);
if ( status ) {
   fprintf (stderr, "Failed to get optimal integer x.\n");
   goto TERMINATE;
}

status = CPXgetmipslack (env, lp, slack, 0, cur_numrows-1);
if ( status ) {
   fprintf (stderr, "Failed to get optimal slack values.\n");
   goto TERMINATE;
}

for (i = 0; i < cur_numrows; i++) {
```

```
      printf ("Row %d:  Slack = %10f\n", i, slack[i]);
   }

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

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

   status = CPXwriteprob (env, lp, "mipex1.lp", NULL);
   if ( status ) {
      fprintf (stderr, "Failed to write LP to disk.\n");
      goto TERMINATE;
   }

TERMINATE:

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

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

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

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

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

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

   /* Free up the problem data arrays, if necessary. */

   free_and_null ((char **) &probname);
   free_and_null ((char **) &obj);
   free_and_null ((char **) &rhs);
   free_and_null ((char **) &sense);
   free_and_null ((char **) &matbeg);
   free_and_null ((char **) &matcnt);
```

```
   free_and_null ((char **) &matind);
   free_and_null ((char **) &matval);
   free_and_null ((char **) &lb);
   free_and_null ((char **) &ub);
   free_and_null ((char **) &ctype);

   return (status);

}  /* END main */


/* This function fills in the data structures for the mixed integer program:

      Maximize
       obj: x1 + 2 x2 + 3 x3 + x4
      Subject To
       c1: - x1 + x2 + x3 + 10x4  <= 20
       c2: x1 - 3 x2 + x3         <= 30
       c3:       x2       - 3.5x4  = 0
      Bounds
       0 <= x1 <= 40
       2 <= x4 <= 3
      Integers
        x4
      End
 */


static int
setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                int *objsen_p, double **obj_p, double **rhs_p,
                char **sense_p, int **matbeg_p, int **matcnt_p,
                int **matind_p, double **matval_p,
                double **lb_p, double **ub_p, char **ctype_p)
{
   char     *zprobname = NULL;     /* Problem name <= 16 characters */
   double   *zobj = NULL;
   double   *zrhs = NULL;
   char     *zsense = NULL;
   int      *zmatbeg = NULL;
   int      *zmatcnt = NULL;
   int      *zmatind = NULL;
   double   *zmatval = NULL;
   double   *zlb = NULL;
   double   *zub = NULL;
   char     *zctype = NULL;
   int      status = 0;

   zprobname = (char *) malloc (16 * sizeof(char));
   zobj      = (double *) malloc (NUMCOLS * sizeof(double));
   zrhs      = (double *) malloc (NUMROWS * sizeof(double));
   zsense    = (char *) malloc (NUMROWS * sizeof(char));
   zmatbeg   = (int *) malloc (NUMCOLS * sizeof(int));
```

```
zmatcnt   = (int *) malloc (NUMCOLS * sizeof(int));
zmatind   = (int *) malloc (NUMNZ * sizeof(int));
zmatval   = (double *) malloc (NUMNZ * sizeof(double));
zlb       = (double *) malloc (NUMCOLS * sizeof(double));
zub       = (double *) malloc (NUMCOLS * sizeof(double));
zctype    = (char *) malloc (NUMCOLS * sizeof(char));

if ( zprobname == NULL || zobj    == NULL ||
     zrhs      == NULL || zsense  == NULL ||
     zmatbeg   == NULL || zmatcnt == NULL ||
     zmatind   == NULL || zmatval == NULL ||
     zlb       == NULL || zub     == NULL ||
     zctype    == NULL                      )  {
   status = 1;
   goto TERMINATE;
}

strcpy (zprobname, "example");

/* The code is formatted to make a visual correspondence
   between the mathematical linear program and the specific data
   items.   */

  zobj[0]  = 1.0;   zobj[1]   = 2.0;  zobj[2]    = 3.0;    zobj[3] = 1.0;

zmatbeg[0] = 0;    zmatbeg[1] = 2;    zmatbeg[2] = 5;   zmatbeg[3] = 7;
zmatcnt[0] = 2;    zmatcnt[1] = 3;    zmatcnt[2] = 2;   zmatcnt[3] = 2;

zmatind[0] = 0;    zmatind[2] = 0;    zmatind[5] = 0;   zmatind[7] = 0;
zmatval[0] = -1.0; zmatval[2] = 1.0;  zmatval[5] = 1.0; zmatval[7] = 10.0;

zmatind[1] = 1;    zmatind[3] = 1;    zmatind[6] = 1;
zmatval[1] = 1.0;  zmatval[3] = -3.0; zmatval[6] = 1.0;

                   zmatind[4] = 2;                     zmatind[8] = 2;
                   zmatval[4] = 1.0;                   zmatval[8] = -3.5;

zlb[0] = 0.0;   zlb[1] = 0.0;      zlb[2] = 0.0;     zlb[3] = 2.0;
zub[0] = 40.0;  zub[1] = CPX_INFBOUND; zub[2] = CPX_INFBOUND; zub[3] = 3.0;

 zctype[0] = 'C';    zctype[1] = 'C';   zctype[2] = 'C';   zctype[3] = 'I';

/* The right-hand-side values don't fit nicely on a line above.  So put
   them here.  */

zsense[0] = 'L';
zrhs[0]   = 20.0;

zsense[1] = 'L';
zrhs[1]   = 30.0;

zsense[2] = 'E';
```

```
   zrhs[2]   = 0.0;

TERMINATE:

   if ( status ) {
      free_and_null ((char **) &zprobname);
      free_and_null ((char **) &zobj);
      free_and_null ((char **) &zrhs);
      free_and_null ((char **) &zsense);
      free_and_null ((char **) &zmatbeg);
      free_and_null ((char **) &zmatcnt);
      free_and_null ((char **) &zmatind);
      free_and_null ((char **) &zmatval);
      free_and_null ((char **) &zlb);
      free_and_null ((char **) &zub);
      free_and_null ((char **) &zctype);
   }
   else {
      *numcols_p   = NUMCOLS;
      *numrows_p   = NUMROWS;
      *objsen_p    = CPX_MAX;   /* The problem is maximization */

      *probname_p  = zprobname;
      *obj_p       = zobj;
      *rhs_p       = zrhs;
      *sense_p     = zsense;
      *matbeg_p    = zmatbeg;
      *matcnt_p    = zmatcnt;
      *matind_p    = zmatind;
      *matval_p    = zmatval;
      *lb_p        = zlb;
      *ub_p        = zub;
      *ctype_p     = zctype;
   }
   return (status);

}  /* END setproblemdata */



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

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */
```

Example: Reading a MIP Problem from a File

This example shows you how to solve a MIP with the Component Libraries when the problem data is stored in a file.

Complete Program: ilomipex2.cpp

This example derives from `ilolpex2.cpp`, an LP example explained in the manual *ILOG CPLEX Getting Started*. That LP example differs from this MIP example in these ways:

- This example solves only MIPs, so it calls only `IloCplex::solve`, and its command line does not require the user to indicate an optimizer.
- This example does not generate or print a basis.

Like other applications based on ILOG CPLEX Concert Technology, this one uses `IloEnv env` to initialize the Concert Technology environment and `IloModel model(env)` to create a problem object. Before it ends, it calls `env.end` to free the environment.

```
// File: examples/src/ilomipex2.cpp
// Version 9.0
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// ilomipex2.cpp - Reading in and optimizing a problem
//
// To run this example, command line arguments are required.
// i.e.,   ilomipex2   filename
//
// Example:
//     ilomipex2  mexample.mps
//

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

static void usage (const char *progname);

int
main (int argc, char **argv)
{
   IloEnv env;
   try {
      IloModel model(env);
      IloCplex cplex(env);
```

```
      if ( argc != 2 ) {
         usage (argv[0]);
         throw(-1);
      }

      IloObjective   obj;
      IloNumVarArray var(env);
      IloRangeArray  rng(env);
      cplex.importModel(model, argv[1], obj, var, rng);

      cplex.extract(model);
      cplex.solve();

      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      env.out() << "Values        = " << vals << endl;
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

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


static void usage (const char *progname)
{
   cerr << "Usage: " << progname << " filename" << endl;
   cerr << "   where filename is a file with extension " << endl;
   cerr << "      MPS, SAV, or LP (lower case is allowed)" << endl;
   cerr << " Exiting..." << endl;
} // END usage
```

Complete Program: mipex2.c

The example derives from `lpex2.c`, an LP example explained in the manual *ILOG CPLEX Getting Started*. That LP example differs from this MIP example in these ways:

- This example solves only MIPs, so it calls only `CPXmipopt`, and its command line does not require the user to indicate an optimizer.

- This example calls `CPXgetstat`, `CPXgetmipobjval`, and `CPXgetmipx` to get a solution. It doesn't generate or print a basis.

Like other applications based on the ILOG CPLEX Callable Library, this one calls `CPXopenCPLEX` to initialize the ILOG CPLEX environment; it sets the screen-indicator parameter to direct output to the screen and calls `CPXcreateprob` to create a problem object. Before it ends, it calls `CPXfreeprob` to free the space allocated to the problem object and `CPXcloseCPLEX` to free the environment.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/mipex2.c                                             */
/*  Version 9.0                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2003 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* mipex2.c - Reading in and optimizing a MIP problem */

/* To run this example, command line arguments are required.
   i.e.,   mipex2   filename
   where
       filename is the name of the file, with .mps, .lp, or .sav extension
   Example:
       mipex2  mexample.mps
 */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

/* Bring in the declarations for the string and character functions
   and malloc */

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

/* Include declarations for functions in this program */

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


int
main (int argc, char *argv[])
{
   /* Declare and allocate space for the variables and arrays where we will
      store the optimization results including the status, objective value,
      and variable values. */
```

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

CPXENVptr     env = NULL;
CPXLPptr      lp = NULL;
int           status;
int           j;
int           cur_numcols;

/* Check the command line arguments */

if ( argc != 2 ) {
   usage (argv[0]);
   goto TERMINATE;
}

/* Initialize the CPLEX environment */

env = CPXopenCPLEX (&status);

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

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

/* Turn on output to the screen */

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

/* Create the problem, using the filename as the problem name */

lp = CPXcreateprob (env, &status, argv[1]);

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

if ( lp == NULL ) {
```

```
   fprintf (stderr, "Failed to create LP.\n");
   goto TERMINATE;
}

/* Now read the file, and copy the data into the created lp */

status = CPXreadcopyprob (env, lp, argv[1], NULL);
if ( status ) {
   fprintf (stderr, "Failed to read and copy the problem data.\n");
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */

status = CPXmipopt (env, lp);

if ( status ) {
   fprintf (stderr, "Failed to optimize MIP.\n");
   goto TERMINATE;
}

solstat = CPXgetstat (env, lp);
printf ("Solution status %d.\n", solstat);

status  = CPXgetmipobjval (env, lp, &objval);

if ( status ) {
   fprintf (stderr,"Failed to obtain objective value.\n");
   goto TERMINATE;
}

printf ("Objective value %.10g\n", objval);

/* The size of the problem should be obtained by asking CPLEX what
   the actual size is. cur_numcols stores the current number
   of columns. */

cur_numcols = CPXgetnumcols (env, lp);

/* Allocate space for solution */

x = (double *) malloc (cur_numcols*sizeof(double));

if ( x == NULL ) {
   fprintf (stderr, "No memory for solution values.\n");
   goto TERMINATE;
}

status = CPXgetmipx (env, lp, x, 0, cur_numcols-1);
if ( status ) {
   fprintf (stderr, "Failed to obtain solution.\n");
   goto TERMINATE;
}

/* Write out the solution */

for (j = 0; j < cur_numcols; j++) {
   printf ( "Column %d:  Value = %17.10g\n", j, x[j]);
```

```
   }

TERMINATE:

   /* Free up the solution */

   free_and_null ((char **) &x);

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

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

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

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

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

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

   return (status);

}  /* END main */


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

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */


static void
usage (char *progname)
{
   fprintf (stderr,"Usage: %s filename\n", progname);
   fprintf (stderr,"   where filename is a file with extension \n");
   fprintf (stderr,"      MPS, SAV, or LP (lower case is allowed)\n");
```

```
   fprintf (stderr,"  This program uses the CPLEX MIP optimizer.\n");
   fprintf (stderr," Exiting...\n");
} /* END usage */
```

Example: Using SOS and Priority

This example illustrates how to use SOS and priority orders.

Complete Program: ilomipex3.cpp

It derives from `ilomipex1.cpp`. The differences between that simpler MIP example and this one are:

- The problem solved is slightly different so the output is interesting. The actual SOS and priority order that the example implements are arbitrary; they do not necessarily represent good data for this problem.
- The routine `setPriorities` sets the SOS and priority order:

```
// File: examples/src/ilomipex3.cpp
// Version 9.0
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// ilomipex3.cpp - Entering and optimizing a MIP problem with SOS sets
//                 and priority orders.  Is a modification of ilomipex1.cpp.
//                 Note that the problem solved is slightly different than
//                 ilomipex1.cpp so that the output is interesting.

#include <ilcplex/ilocplex.h>

ILOSTLBEGIN

static void
   populatebyrow(IloModel model, IloNumVarArray var, IloRangeArray con);

int
main (void) {
   IloEnv env;
   try {
      IloModel model(env);

      IloNumVarArray var(env);
```

```
      IloRangeArray con(env);
      populatebyrow (model, var, con);

      IloCplex cplex(model);
      IloNumVarArray ordvar(env, 2, var[1], var[3]);
      IloNumArray    ordpri(env, 2, 8.0, 7.0);
      cplex.setPriorities (ordvar, ordpri);
      cplex.setDirection(var[1], IloCplex::BranchUp);
      cplex.setDirection(var[3], IloCplex::BranchDown);
      cplex.solve();

      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      env.out() << "Values        = " << vals << endl;
      cplex.getSlacks(vals, con);
      env.out() << "Slacks        = " << vals << endl;

      cplex.exportModel("mipex3.lp");
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();
   return 0;

}  // END main


static void
populatebyrow (IloModel model, IloNumVarArray x, IloRangeArray c)
{
   IloEnv env = model.getEnv();

   x.add(IloNumVar(env, 0.0, 40.0));
   x.add(IloNumVar(env, 0.0, IloInfinity, ILOINT));
   x.add(IloNumVar(env, 0.0, IloInfinity, ILOINT));
   x.add(IloNumVar(env, 2.0, 3.0, ILOINT));
   model.add(IloMaximize(env, x[0] + 2 * x[1] + 3 * x[2] + x[3]));

   c.add( - x[0] +     x[1] + x[2] + 10 * x[3] <= 20);
   c.add(   x[0] - 3 * x[1] + x[2]             <= 30);
   c.add(              x[1]        - 3.5* x[3] == 0);
   model.add(c);

   model.add(IloSOS1(model.getEnv(),
                     IloNumVarArray(model.getEnv(), 2, x[2], x[3]),
```

```
                        IloNumArray(env, 2, 25.0, 18.0)    ));

}  // END populatebyrow
```

Complete Program: mipex3.c

This example derives from `mipex1.c`. The differences between that simpler MIP example and this one are:

- The problem solved is slightly different so the output is interesting. The actual SOS and priority order that the example implements are arbitrary; they do not necessarily represent good data for this problem.
- The routine `CPXwriteprob` writes the problem to disk before the example copies the SOS and priority order to verify that the base problem was copied correctly.
- The ILOG CPLEX preprocessing parameters for the presolver and aggregator are turned off to make the output interesting. Generally, this is not required nor recommended.
- The routine `setsosandorder` sets the SOS and priority order:
  - It calls `CPXcopysos` to copy the SOS into the problem object.
  - It calls `CPXcopyorder` to copy the priority order into the problem object.
  - It writes the SOS information to files by calling `CPXsoswrite`.
  - It writes the priority order to files by calling `CPXordwrite`.


```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/mipex3.c                                             */
/*  Version 9.0                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2003 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* mipex3.c - Entering and optimizing a MIP problem with SOS sets
              and priority orders.  Is a modification of mipex1.c .
              Note that the problem solved is slightly different than
              mipex1.c so that the output is interesting.  */

/* Bring in the CPLEX function declarations and the C library
   header file stdlib.h with the following single include. */

#include <ilcplex/cplex.h>

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

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

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

static int
   setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                   int *objsen_p, double **obj_p, double **rhs_p,
                   char **sense_p, int **matbeg_p, int **matcnt_p,
                   int **matind_p, double **matval_p,
                   double **lb_p, double **ub_p, char **ctype_p),
   setsosandorder (CPXENVptr env, CPXLPptr lp);

static void
   free_and_null (char **ptr);



/* The problem we are optimizing will have 3 rows, 4 columns
   and 9 nonzeros.  */

#define NUMROWS    3
#define NUMCOLS    4
#define NUMNZ      9


int
main (void)
{
   /* Declare and allocate space for the variables and arrays that
      will contain the data which define the LP problem */

   char     *probname = NULL;
   int      numcols;
   int      numrows;
   int      objsen;
   double   *obj = NULL;
   double   *rhs = NULL;
   char     *sense = NULL;
   int      *matbeg = NULL;
   int      *matcnt = NULL;
   int      *matind = NULL;
   double   *matval = NULL;
   double   *lb = NULL;
   double   *ub = NULL;
   char     *ctype = NULL;

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

   int      solstat;
   double   objval;
```

```
double   x[NUMCOLS];
double   slack[NUMROWS];


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

/* Initialize the CPLEX environment */

env = CPXopenCPLEX (&status);

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

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

/* Turn on output to the screen */

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

/* Fill in the data for the problem.  */

status = setproblemdata (&probname, &numcols, &numrows, &objsen, &obj,
                         &rhs, &sense, &matbeg, &matcnt, &matind, &matval,
                         &lb, &ub, &ctype);
if ( status ) {
   fprintf (stderr, "Failed to build problem data arrays.\n");
   goto TERMINATE;
}

/* Create the problem. */

lp = CPXcreateprob (env, &status, probname);
```

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

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

/* Now copy the problem data into the lp */

status = CPXcopylp (env, lp, numcols, numrows, objsen, obj, rhs,
                    sense, matbeg, matcnt, matind, matval,
                    lb, ub, NULL);

if ( status ) {
   fprintf (stderr, "Failed to copy problem data.\n");
   goto TERMINATE;
}

/* Now copy the ctype array */

status = CPXcopyctype (env, lp, ctype);
if ( status ) {
   fprintf (stderr, "Failed to copy ctype\n");
   goto TERMINATE;
}

/* Write a copy of the problem to a file. */

status = CPXwriteprob (env, lp, "mipex3.mps", NULL);
if ( status ) {
   fprintf (stderr, "Failed to write LP to disk.\n");
   goto TERMINATE;
}

/* Set up the SOS set and priority order */

status = setsosandorder (env, lp);
if ( status ) goto TERMINATE;

/* Turn off CPLEX presolve, aggregate, and print every node.  This
   is just to make it interesting.  Turning off CPLEX presolve is
   NOT recommended practice !! */

status = CPXsetintparam (env, CPX_PARAM_PREIND, CPX_OFF);
if (!status) CPXsetintparam (env, CPX_PARAM_AGGIND, CPX_OFF);
if (!status) CPXsetintparam (env, CPX_PARAM_MIPINTERVAL, 1);

if ( status ) {
```

```
   fprintf (stderr, "Failed to set some CPLEX parameters.\n");
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */

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

solstat = CPXgetstat (env, lp);

/* Write the output to the screen. */

printf ("\nSolution status = %d\n", solstat);

status = CPXgetmipobjval (env, lp, &objval);
if ( status ) {
   fprintf (stderr,"No MIP objective value available.  Exiting...\n");
   goto TERMINATE;
}

printf ("Solution value  = %f\n\n", objval);

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

cur_numrows = CPXgetnumrows (env, lp);
cur_numcols = CPXgetnumcols (env, lp);

status = CPXgetmipx (env, lp, x, 0, cur_numcols-1);
if ( status ) {
   fprintf (stderr, "Failed to get optimal integer x.\n");
   goto TERMINATE;
}

status = CPXgetmipslack (env, lp, slack, 0, cur_numrows-1);
if ( status ) {
   fprintf (stderr, "Failed to get optimal slack values.\n");
   goto TERMINATE;
}

for (i = 0; i < cur_numrows; i++) {
   printf ("Row %d:  Slack = %10f\n", i, slack[i]);
}

for (j = 0; j < cur_numcols; j++) {
   printf ("Column %d:  Value = %10f\n", j, x[j]);
```

```
   }

TERMINATE:

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

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

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

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

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

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

   /* Free up the problem data arrays, if necessary. */

   free_and_null ((char **) &probname);
   free_and_null ((char **) &obj);
   free_and_null ((char **) &rhs);
   free_and_null ((char **) &sense);
   free_and_null ((char **) &matbeg);
   free_and_null ((char **) &matcnt);
   free_and_null ((char **) &matind);
   free_and_null ((char **) &matval);
   free_and_null ((char **) &lb);
   free_and_null ((char **) &ub);
   free_and_null ((char **) &ctype);

   return (status);

}  /* END main */


/* This function fills in the data structures for the mixed integer program:

      Maximize
```

```
       obj: x1 + 2 x2 + 3 x3 + x4
      Subject To
       c1: - x1 + x2 + x3 + 10x4  <= 20
       c2: x1 - 3 x2 + x3         <= 30
       c3:       x2       - 3.5x4  = 0
      Bounds
       0 <= x1 <= 40
       2 <= x4 <= 3
      Integers
        x2 x3 x4
      End
 */


static int
setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                int *objsen_p, double **obj_p, double **rhs_p,
                char **sense_p, int **matbeg_p, int **matcnt_p,
                int **matind_p, double **matval_p,
                double **lb_p, double **ub_p, char **ctype_p)
{
   char     *zprobname = NULL;     /* Problem name <= 16 characters */
   double   *zobj = NULL;
   double   *zrhs = NULL;
   char     *zsense = NULL;
   int      *zmatbeg = NULL;
   int      *zmatcnt = NULL;
   int      *zmatind = NULL;
   double   *zmatval = NULL;
   double   *zlb = NULL;
   double   *zub = NULL;
   char     *zctype = NULL;
   int      status = 0;

   zprobname = (char *) malloc (16 * sizeof(char));
   zobj      = (double *) malloc (NUMCOLS * sizeof(double));
   zrhs      = (double *) malloc (NUMROWS * sizeof(double));
   zsense    = (char *) malloc (NUMROWS * sizeof(char));
   zmatbeg   = (int *) malloc (NUMCOLS * sizeof(int));
   zmatcnt   = (int *) malloc (NUMCOLS * sizeof(int));
   zmatind   = (int *) malloc (NUMNZ * sizeof(int));
   zmatval   = (double *) malloc (NUMNZ * sizeof(double));
   zlb       = (double *) malloc (NUMCOLS * sizeof(double));
   zub       = (double *) malloc (NUMCOLS * sizeof(double));
   zctype    = (char *) malloc (NUMCOLS * sizeof(char));

   if ( zprobname == NULL || zobj    == NULL ||
        zrhs      == NULL || zsense  == NULL ||
        zmatbeg   == NULL || zmatcnt == NULL ||
        zmatind   == NULL || zmatval == NULL ||
        zlb       == NULL || zub     == NULL ||
        zctype    == NULL                     )  {
```

```
      status = 1;
      goto TERMINATE;
   }

   strcpy (zprobname, "example");

   /* The code is formatted to make a visual correspondence
      between the mathematical linear program and the specific data
      items.   */

     zobj[0] = 1.0;   zobj[1]   = 2.0;  zobj[2] = 3.0;    zobj[3] = 1.0;

   zmatbeg[0] = 0;    zmatbeg[1] = 2;   zmatbeg[2] = 5;   zmatbeg[3] = 7;
   zmatcnt[0] = 2;    zmatcnt[1] = 3;   zmatcnt[2] = 2;   zmatcnt[3] = 2;

   zmatind[0] = 0;    zmatind[2] = 0;   zmatind[5] = 0;   zmatind[7] = 0;
   zmatval[0] = -1.0; zmatval[2] = 1.0; zmatval[5] = 1.0; zmatval[7] = 10.0;

   zmatind[1] = 1;    zmatind[3] = 1;    zmatind[6] = 1;
   zmatval[1] = 1.0;  zmatval[3] = -3.0; zmatval[6] = 1.0;

                      zmatind[4] = 2;                    zmatind[8] = 2;
                      zmatval[4] = 1.0;                  zmatval[8] = -3.5;

   zlb[0] = 0.0;    zlb[1] = 0.0;          zlb[2] = 0.0;          zlb[3] = 2.0;
   zub[0] = 40.0;   zub[1] = CPX_INFBOUND; zub[2] = CPX_INFBOUND; zub[3] = 3.0;

   zctype[0] = 'C';   zctype[1] = 'I';  zctype[2] = 'I';   zctype[3] = 'I';

  /* The right-hand-side values don't fit nicely on a line above.  So put
     them here.  */

   zsense[0] = 'L';
   zrhs[0]   = 20.0;

   zsense[1] = 'L';
   zrhs[1]   = 30.0;

   zsense[2] = 'E';
   zrhs[2]   = 0.0;

TERMINATE:

   if ( status ) {
      free_and_null ((char **) &zprobname);
      free_and_null ((char **) &zobj);
      free_and_null ((char **) &zrhs);
      free_and_null ((char **) &zsense);
      free_and_null ((char **) &zmatbeg);
      free_and_null ((char **) &zmatcnt);
      free_and_null ((char **) &zmatind);
      free_and_null ((char **) &zmatval);
      free_and_null ((char **) &zlb);
```

```
      free_and_null ((char **) &zub);
      free_and_null ((char **) &zctype);
   }
   else {
      *numcols_p   = NUMCOLS;
      *numrows_p   = NUMROWS;
      *objsen_p    = CPX_MAX;   /* The problem is maximization */

      *probname_p  = zprobname;
      *obj_p       = zobj;
      *rhs_p       = zrhs;
      *sense_p     = zsense;
      *matbeg_p    = zmatbeg;
      *matcnt_p    = zmatcnt;
      *matind_p    = zmatind;
      *matval_p    = zmatval;
      *lb_p        = zlb;
      *ub_p        = zub;
      *ctype_p     = zctype;
   }
   return (status);

}  /* END setproblemdata */


static int
setsosandorder (CPXENVptr env, CPXLPptr lp)
{
   /* Priority order information */
   int  colindex[2];
   int  priority[2];
   int  direction[2];

   /* SOS set information */
   char   sostype[1];
   int    sospri[1];
   int    sosbeg[1];
   int    sosind[2];
   double sosref[2];

   int  status = 0;

   /* Note - for this example, the priority order and SOS information
      are just made up for illustrative purposes.  The priority order
      is not necessarily a good one for this particular problem.  */

   /* Set order info.  Variables 1 and 3 will be in the priority order,
      with respective priorities of 8 and 7, and with respective
      branching directions of up and down */

   colindex[0]  = 1;               colindex[1]  = 3;
   priority[0]  = 8;               priority[1]  = 7;
```

```
   direction[0] = CPX_BRANCH_UP;     direction[1] = CPX_BRANCH_DOWN;

   status = CPXcopyorder (env, lp, 2, colindex, priority, direction);
   if ( status ) {
      fprintf (stderr, "CPXcopyorder failed.\n");
      goto TERMINATE;
   }

   /* Set SOS set info.  Create one SOS type 1 set, with variables
      2 and 3 in it, with set priority 3, and reference values
      25 and 18  for the 2 variables, respectively.   */

   sostype[0] = CPX_TYPE_SOS1;
   sospri[0]  = 3;
   sosbeg[0]  = 0;
   sosind[0]  = 2;    sosind[1] = 3;
   sosref[0]  = 25;   sosref[1] = 18;

   status = CPXcopysos (env, lp, 1, 2, sostype, sospri,
                        sosbeg, sosind, sosref);
   if ( status ) {
      fprintf (stderr, "CPXcopysos failed.\n");
      goto TERMINATE;
   }

   /* To assist in debugging, write the order and SOS to a file. */

   status = CPXordwrite (env, lp, "mipex3.ord");
   if ( status ) {
      fprintf (stderr, "CPXordwrite failed.\n");
      goto TERMINATE;
   }

   status = CPXsoswrite (env, lp, "mipex3.sos");
   if ( status ) {
      fprintf (stderr, "CPXsoswrite failed.\n");
      goto TERMINATE;
   }

TERMINATE:

   return (status);

}



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

static void
free_and_null (char **ptr)
{
```

```
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */
```

C H A P T E R

14

Cutting Stock: Column Generation

This chapter uses an example of cutting stock to demonstrate the technique of column generation in Concert Technology. In it, you will learn:

- how to use classes of ILOG CPLEX for column generation in column-wise modeling;
- how to modify a model and re-optimize;
- how to change the type of a variable with `IloConversion`;
- how to use more than one model;
- how to use more than one algorithm (instances of `IloCplex`, for example).

What Is Column Generation?

In colloquial terms, column generation is a way of beginning with a small, manageable *part* of a problem (specifically, a few of the variables), solving that part, analyzing that partial solution to determine the next part of the problem (specifically, one or more variables) to add to the model, and then resolving the enlarged model. Column-wise modeling repeats that process until it achieves a satisfactory solution to the whole of the problem.

In formal terms, column generation is a way of solving a linear programming problem that adds columns (corresponding to constrained variables) during the pricing phase of the simplex method of solving the problem. In gross terms, generating a column in the primal

simplex formulation of a linear programming problem corresponds to adding a constraint in its dual formulation. In the dual formulation of a given linear programming problem, you might think of column generation as a cutting plane method.

In that context, many researchers have observed that column generation is a very powerful technique for solving a wide range of industrial problems to optimality or to near optimality. Ford and Fulkerson, for example, suggested column generation in the context of a multi-commodity network flow problem as early as 1958 in the journal of *Management Science*. By 1960, Dantzig and Wolfe had adapted it to linear programming problems with a decomposable structure. Gilmore and Gomory then demonstrated its effectiveness in a cutting stock problem. More recently, vehicle routing, crew scheduling, and other integer-constrained problems have motivated further research into column generation.

Column generation rests on the fact that in the simplex method, the solver does not need access to all the variables of the problem simultaneously. In fact, a solver can begin work with only the basis (a particular subset of the constrained variables) and then use reduced cost to determine which other variables to access as needed.

Column-Wise Models in Concert Technology

Concert Technology offers facilities for exploiting column generation. In particular, you can design the model of your problem (one or more instances of the class `IloModel`) in terms of a basis and added columns (instances of `IloNumVar`, `IloNumVarArray`, `IloNumColumn`, or `IloNumColumnArray`). For example, instances of `IloNumColumn` represent *columns*, and you can use `operator()` in the classes `IloObjective` and `IloRange` to create *terms* in column expressions. In practice, the column serves as a kind of place holder for a variable in other extractable objects (such as a range constraint or an objective) when your application needs to declare or use those other extractable objects before it can actually know the value of a variable appearing in them.

Furthermore, an instance of `IloCplex` provides a way to solve the master linear problem, while other Concert Technology algorithms (that is, instances of `IloSolver`, of `IloCplex` itself, or of subclasses of `IloAlgorithm`, for example) lend themselves to other parts of the problem by determining which variables to consider next (and thus which columns to generate).

In the *Concert Technology Reference Manual*, the concept *Column-Wise Modeling* provides more detail about this topic and offers simple examples of its use.

Describing the Problem

The cutting stock problem in this chapter is sometimes known in math programming terms as a knapsack problem with reduced cost in the objective function.

Generally, a cutting stock problem begins with a supply of rolls of material (the stock). Strips are cut from a continuous roll. All the strips cut from one roll are known together as a *pattern*. The point of this example is to use as few rolls of stock as possible to satisfy some specified demand. By convention, it is assumed that only one pattern is laid out across the stock; consequently, only one dimension—the width—of each roll of stock is important.

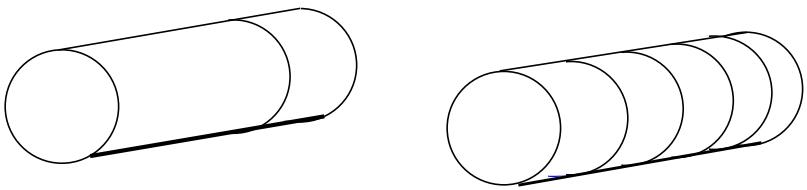

Figure 14.1 *Two different patterns from a roll of stock*

Even with that simplifying assumption, the fact that there can be so many different patterns makes a naive model of this problem (where a user declares one variable for every possible pattern) impractical. Such a model introduces too many symmetries. Fortunately, for any given customer order, a limited number of patterns will suffice, so many of the possible patterns can be disregarded, and the application can focus on finding the relevant ones.

Here is a conventional statement of a cutting stock problem in terms of the unknown X_j, the number of times that pattern j will be used, and A_{ij}, the number of items i of each pattern j needed to satisfy demand d_i:

Minimize $\sum_j X_j$

subject to $\sum_{ij} A_{ij} X_j$ with $X_j \geq 0$

Solving this model with all columns present from the beginning is practically impossible. In fact, even with only 10 types of items with a size roughly 1/10 of the width of the roll, there would exist roughly 10^10 kinds of patterns, and hence that many decision variables. Such a formulation might not even fit in memory on a reasonably large computer. Moreover, most of those patterns would obviously not be interesting in a solution. These considerations make column generation an interesting approach for this problem.

To solve a cutting stock problem by column generation, start with a subproblem. Choose one pattern, lay it out on the stock, and cut as many items as possible, subject to the constraints of demand for that item and the width of the stock. This procedure will surely work in that it produces some answer (a feasible solution) to the problem, but it will not necessarily produce a satisfactory answer in this way.

To move closer to a satisfactory solution, the application can then generate other columns. That is, other decision variables (other X_j) will be chosen to add to the model. Those decision variables are chosen on the basis of their favorable reduced cost. In linear programming terms, choose variables for which the negative reduced cost improves the minimization. In this formulation of the rest of the problem, π_i represents the dual value of constraint *i*, like this:

Minimize $1 - \sum_i \pi_i A_i$

subject to $\sum_i W_i A_i \leq W$

where W is the width of a roll of stock and A_i must be a nonnegative integer.

Representing the Data

As usual in a Concert Technology application, an environment, an instance of `IloEnv`, is created first to organize the data and build the model of the problem.

The data defining this problem includes the width of a roll of stock.This value is read from a file and represented by a numeric value, `rollWidth`. The widths of the ordered rolls are also read from a file and put into an array of numeric values, `size`. Finally, the number of rolls ordered of each width is read from a file and put into an array of numeric values, `amount`.

Developing the Model: Building and Modifying

In this problem, an initial model `cutOpt` is built first. Later, through its modifications, another model `patGen` is built to generate the new columns.

The first model `cutOpt`, an instance of `IloModel`, is declared like this:

```
IloModel cutOpt (env);
```

An array of numeric variables (an instance of `IloNumVarArray`) `Cut[j]` is declared to represent the decision variables in the model. The index `j` indicates a pattern using the widths found in the array `size`. Consequently, a given variable `Cut[j]` in the solution will be the number of rolls to cut according to pattern `j`.

As a model for this problem is built, there will be opportunities to demonstrate to you how to modify a model by adding extractable objects, adding columns, changing coefficients in an objective function, and changing the type of a variable. When you modify a model by means

of the methods of extractable objects, Concert Technology notifies the algorithms (instances of subclasses of `IloAlgorithm`, such as `IloCplex` or `IloSolver`) about the modification. (For more about that idea, see the concept of *Notification* in the *Concert Technology Reference Manual*.)

When `IloCplex`, for example, is notified about a change in an extractable object that it has extracted, it maintains as much of the current solution information as it can accurately and reasonably. Other parts of the *ILOG CPLEX User's Manual* offer more detail about how the algorithm responds to modifications in the model.

Adding Extractable Objects: Both Ways

In a Concert Technology application, there are two ways of adding extractable objects to a model: by means of a template function (`IloAdd`) or by means of a method of the model (`IloModel::add` or `IloArray::add`). In this example, you see both ways.

Using a Template to Add Objects

When an objective is added to the model, the application needs to keep a handle to the objective `RollsUsed` because it is needed when the application generates columns. For that purpose, the application relies on the template function `IloAdd`, like this:

```
IloObjective   RollsUsed = IloAdd(cutOpt, IloMinimize(env));
```

Apart from the fact that it preserves type information, that single line is equivalent to these lines:

```
IloObjective RollsUsed = IloMinimize(env);
cutOpt.add(RollsUsed);
```

Likewise, the application adds an array of constraints to the model. These constraints are needed later in column generation as well, so the application again uses `IloAdd` again to add the array `Fill` to the model.

```
IloRangeArray  Fill = IloAdd(cutOpt,
                             IloRangeArray(env, amount, IloInfinity));
```

That statement creates `amount.getSize` range constraints. Constraint `Fill[i]` has a lower bound of `amount[i]` and an upper bound of `IloInfinity`.

Using a Method to Add Objects

It is also possible to add objects to your model by means of the method `IloModel::add`. This example uses that approach in these lines:

```
patGen.add(IloScalProd(size, Use) <= rollWidth);
```

In contrast, to create `Cut`, an array of decision variables, the application uses the `add` method of `IloArray` within a loop, like this:

```
IloInt nWdth = size.getSize();
for (j = 0; j < nWdth; j++)
  Cut.add(IloNumVar(RollsUsed(1) + Fill[j](int(rollWidth / size[j]))));
```

In that loop, the variable `Cut[j]` is created with an objective coefficient of 1 (one), and all its pattern coefficients are initially 0 (zero) except for the one in `Fill[j]`. That one is set to `(int(rollWidth / size[j]))`. By default, the type of each variable is `IloFloat`, so it is not explicitly mentioned in the loop.

Adding Columns to a Model

Creating a new column to add to a model in Concert Technology is a two-step process:

1. Create a column *expression* defining the new column.
2. Create a *variable* using that column expression and add the variable to the model.

For example, in this problem, `RollsUsed` is an instance of `IloObjective`. The statement `RollsUsed(1)` creates a *term* in a column *expression* defining how to add a new *variable* as a linear term with a coefficient of 1 (one) to the expression `RollsUsed`.

The terms of a column expression are connected to one another by the overloaded `operator +`.

The constraints of this problem are represented in the array `Fill`. A new column to be added to the model has a coefficient for each of the constraints in `Fill`. Those coefficients are represented in the array `newPatt`.

Changing Coefficients of an Objective Function

According to that two-step procedure for adding a column to a model, the following lines create the column with coefficient 1 (one) for the objective `RollsUsed` and create the new variable with bounds at 0 (zero) and at `MAXCUT`.

```
IloNumColumn col = RollsUsed(1);
for (IloInt i = 0; i < Fill.getSize(); ++i)
     col += Fill[i](newPatt[i]);
IloNumVar var(col, 0, MAXCUT);
```

(However, those lines do not appear in the example at hand.) Concert Technology offers a shortcut in the `operator()` for an array of range constraints. Those lines of code can be condensed into the following line:

```
IloNumVar var(RollsUsed(1) + Fill(newPatt), 0, MAXCUT);
```

In other words, `Fill(newPatt)` returns the column expression that the loop would create. You will see a similar shortcut in the example.

There are also other ways of modifying an objective in a Concert Technology model. For example, you can call the method `IloObjective::setCoef` to change a *coefficient* in an objective. In this example, you see this way of changing a coefficient when the application wants to find a new pattern.

```
ReducedCost.setCoef(Use, price);
```

You can also change the *sense* of an objective in Concert Technology. There are at least two ways to do so. One approach uses the method `setSense` of the class `IloObjective`. The other approach first removes the current objective function from the model, then uses either the function `IloMinimize` or `IloMaximize` to create a new objective function, and finally adds that new objective to the model.

Changing the Type of a Variable

With Concert Technology, in order to change the type of a variable in a model, you actually create an extractable object (an instance of `IloConversion`, documented in the *Concert Technology Reference Manual*) and add that object to the model.

In the example, when the application needs to change the elements of `Cut` (an array of numeric variables) from their default type of `ILOFLOAT` to integer (type `ILOINT`), it creates an instance of `IloConversion`, applies it the array `Cut`, and adds the conversion to the model, `cutOpt`, like this:

```
cutOpt.add(IloConversion(env, Cut, ILOINT));
```

Cut Optimization Model

Here is a summary of the initial model `cutOpt`:

```
IloModel cutOpt (env);

IloObjective   RollsUsed = IloAdd(cutOpt, IloMinimize(env));
IloRangeArray  Fill = IloAdd(cutOpt,
                             IloRangeArray(env, amount, IloInfinity));
IloNumVarArray Cut(env);

IloInt nWdth = size.getSize();
for (j = 0; j < nWdth; j++)
  Cut.add(IloNumVar(RollsUsed(1) + Fill[j](int(rollWidth / size[j]))));
```

Pattern Generator Model

The application in this example starts with an initial model `cutOpt` and through modifications eventually builds a second model `patGen`. This pattern generator `patGen` (in contrast to `cutOpt`) includes integer variables in the array `Use`. That array appears in the only constraint added to `patGen`: a scalar product making sure that the patterns used do not exceed the width of rolls. The application also adds a rudimentary objective function to

patGen. This objective initially consists of only the constant 1 (one). The rest of the objective function depends on the solution found with the initial model cutOpt. The application will build that objective function as that information is computed. Here, in short, is patGen:

```
IloModel patGen (env);

IloObjective ReducedCost = IloAdd(patGen, IloMinimize(env, 1));
IloNumVarArray Use(env, nWdth, 0, IloInfinity, ILOINT);
patGen.add(IloScalProd(size, Use) <= rollWidth);
```

Solving the Problem: Using More than One Algorithm

This example does not solve the problem to optimality. It only generates a good feasible solution. It does so by first solving a linear relaxation of the problem. In other words, the decision variables Cut[j] are relaxed (not required to be integers). This linear relaxation of the problem then generates columns. While generating columns, the application keeps the variables generated so far, changes their type to integer, and resolves the problem.

As you've seen, this example effectively builds two models of the problem, cutOpt and patGen. Likewise, it uses two algorithms (that is, two instances of IloCplex) to arrive at a good feasible solution.

Here's how to create the first algorithm cutSolver and extract the initial model cutOpt:

```
IloCplex cutSolver(cutOpt);
```

And here is how to create the second algorithm and extract patGen:

```
IloCplex patSolver(patGen);
```

The heart of the example is here, in the column generation and optimization over current patterns:

```
IloNumArray price(env, nWdth);
IloNumArray newPatt(env, nWdth);

for (;;) {
  /// OPTIMIZE OVER CURRENT PATTERNS ///

  cutSolver.solve();
  report1 (cutSolver, Cut, Fill);

  /// FIND AND ADD A NEW PATTERN ///

  for (i = 0; i < nWdth; i++)
    price[i] = -cutSolver.getDual(Fill[i]);
  ReducedCost.setCoef(Use, price);

  patSolver.solve();
  report2 (patSolver, Use, ReducedCost);

  if (patSolver.getValue(ReducedCost) > -RC_EPS) break;

  patSolver.getValues(newPatt, Use);
  Cut.add( IloNumVar(RollsUsed(1) + Fill(newPatt)) );
}
cutOpt.add(IloConversion(env, Cut, ILOINT));
cutSolver.solve();
```

Those lines solve the current subproblem `cutOpt` by calling `cutSolver.solve`. Then they copy the values of the negative dual solution into the array `price`. They use that array to set objective coefficients in the model `patGen`. Then they solve the right pattern generation problem.

If the objective value exceeds the negation of the optimality tolerance (`-RC_EPS`), then the application has proved that the current solution of the model `cutOpt` is optimal within the given optimality tolerance (`RC_EPS`). Otherwise, the application copies the solution of the current pattern generation problem into the array `newPatt` and uses that new pattern to build the next column to add to the model `cutOpt`. Then it repeats the procedure.

Ending the Program

As in other C++ Concert Technology applications, this program ends with a call to `IloEnv::end` to de-allocate the models and algorithms once they are no longer in use.

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

Complete Program

You can see the entire program here or online in the standard distribution of ILOG CPLEX in the file `/examples/src/cutstock.cpp`. To run the example, you need only a license for ILOG CPLEX. It is also possible to write your own pattern generation models to use with algorithms of ILOG Solver and ILOG Hybrid Optimizers. In that case, you need a license for those products.

```
// ------------------------------------------------------------- -*- C++ -*-
// File: examples/src/cutstock.cpp
// Version 9.0
// ------------------------------------------------------------------------
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ------------------------------------------------------------------------
//

#include <ilcplex/ilocplex.h>

ILOSTLBEGIN

#define RC_EPS 1.0e-6

typedef IloArray<IloNumArray> IloNumArray2;


static void readData (const char* filename, IloNum& rollWidth,
                      IloNumArray& size, IloNumArray& amount);
static void report1 (IloCplex& cutSolver, IloNumVarArray Cut, IloRangeArray Fill);
static void report2 (IloAlgorithm& patSolver,
                     IloNumVarArray Use,
                     IloObjective obj);
static void report3 (IloCplex& cutSolver, IloNumVarArray Cut);



/// MAIN PROGRAM ///

int
main(int argc, char **argv)
{
   IloEnv env;
   try {
      IloInt  i, j;

      IloNum      rollWidth;
      IloNumArray amount(env);
      IloNumArray size(env);

      if ( argc > 1 )
```

```
   readData(argv[1], rollWidth, size, amount);
else
   readData("../../../examples/data/cutstock.dat", rollWidth, size, amount);

/// CUTTING-OPTIMIZATION PROBLEM ///

IloModel cutOpt (env);

IloObjective   RollsUsed = IloAdd(cutOpt, IloMinimize(env));
IloRangeArray  Fill = IloAdd(cutOpt, IloRangeArray(env, amount, IloInfinity));
IloNumVarArray Cut(env);

IloInt nWdth = size.getSize();
for (j = 0; j < nWdth; j++) {
   Cut.add(IloNumVar(RollsUsed(1) + Fill[j](int(rollWidth / size[j]))));
}

IloCplex cutSolver(cutOpt);

/// PATTERN-GENERATION PROBLEM ///

IloModel patGen (env);

IloObjective ReducedCost = IloAdd(patGen, IloMinimize(env, 1));
IloNumVarArray Use(env, nWdth, 0, IloInfinity, ILOINT);
patGen.add(IloScalProd(size, Use) <= rollWidth);

IloCplex patSolver(patGen);

/// COLUMN-GENERATION PROCEDURE ///

IloNumArray price(env, nWdth);
IloNumArray newPatt(env, nWdth);

/// COLUMN-GENERATION PROCEDURE ///

for (;;) {
   /// OPTIMIZE OVER CURRENT PATTERNS ///

   cutSolver.solve();
   report1 (cutSolver, Cut, Fill);

   /// FIND AND ADD A NEW PATTERN ///

   for (i = 0; i < nWdth; i++) {
     price[i] = -cutSolver.getDual(Fill[i]);
   }
   ReducedCost.setCoef(Use, price);

   patSolver.solve();
   report2 (patSolver, Use, ReducedCost);

   if (patSolver.getValue(ReducedCost) > -RC_EPS) break;
```

```
            patSolver.getValues(newPatt, Use);
            Cut.add( IloNumVar(RollsUsed(1) + Fill(newPatt)) );
         }

         cutOpt.add(IloConversion(env, Cut, ILOINT));

         cutSolver.solve();
         report3 (cutSolver, Cut);
      }
      catch (IloException& ex) {
         cerr << "Error: " << ex << endl;
      }
      catch (...) {
         cerr << "Error" << endl;
      }

      env.end();

      return 0;
   }


   static void readData (const char* filename, IloNum& rollWidth,
                         IloNumArray& size, IloNumArray& amount)
   {
      ifstream in(filename);
      if (in) {
         in >> rollWidth;
         in >> size;
         in >> amount;
      }
      else {
         cerr << "No such file: " << filename << endl;
         throw(1);
      }
   }

   static void report1 (IloCplex& cutSolver, IloNumVarArray Cut, IloRangeArray Fill)
   {
      cout << endl;
      cout << "Using " << cutSolver.getObjValue() << " rolls" << endl;
      cout << endl;
      for (IloInt j = 0; j < Cut.getSize(); j++) {
         cout << "  Cut" << j << " = " << cutSolver.getValue(Cut[j]) << endl;
      }
      cout << endl;
      for (IloInt i = 0; i < Fill.getSize(); i++) {
         cout << "  Fill" << i << " = " << cutSolver.getDual(Fill[i]) << endl;
      }
      cout << endl;
   }

   static void report2 (IloAlgorithm& patSolver, IloNumVarArray Use, IloObjective obj)
```

```
{
   cout << endl;
   cout << "Reduced cost is " << patSolver.getValue(obj) << endl;
   cout << endl;
   if (patSolver.getValue(obj) <= -RC_EPS) {
      for (IloInt i = 0; i < Use.getSize(); i++)  {
         cout << "  Use" << i << " = " << patSolver.getValue(Use[i]) << endl;
      }
      cout << endl;
   }
}

static void report3 (IloCplex& cutSolver, IloNumVarArray Cut)
{
   cout << endl;
   cout << "Best integer solution uses "
        << cutSolver.getObjValue() << " rolls" << endl;
   cout << endl;
   for (IloInt j = 0; j < Cut.getSize(); j++) {
      cout << "  Cut" << j << " = " << cutSolver.getValue(Cut[j]) << endl;
   }
}
```

15

Rates: Using Semi-Continuous Variables

This chapter uses an example of managing production in a power plant to demonstrate semi-continuous variables in Concert Technology. In it, you will learn:

- more about `IloCplex`, the C++ class giving you access to ILOG CPLEX;
- more about building a model with ILOG CPLEX classes;
- how to use the class `IloSemiContVar`.

What Is IloCplex?

The class `IloCplex`, documented in the *ILOG CPLEX Reference Manual*, derives from the class `IloAlgorithm`, documented in the *ILOG Concert Reference Manual*. An instance of `IloCplex` is capable of solving optimization problems known as mixed integer programs (MIPs) and linear programs (LPs).

IloCplex and Extraction

As an ILOG Concert algorithm, an instance of `IloCplex` extracts the following classes from a model when you call its method `IloCplex::extract`:

- `IloNumVar` representing numeric variables;

- `IloSemiContVar` representing semi-continuous or semi-integer variables;
- `IloObjective` representing at most one objective function in linear or piecewise linear expressions;
- `IloRange` representing range constraints in linear or piecewise linear expressions;
- `IloConversion` representing type conversions of variables (from floating-point to integer, for example);
- `IloSOS1` representing special ordered sets of type 1;
- `IloSOS2` representing special ordered sets of type 2;
- `IloAnd` for use with `IloSolution`.

There are also facilities in `IloCplex` for representing piecewise-linear functions, as explained in *Transport: Piecewise Linear Optimization* on page 397, and logical constraints, introduced in *Logical Constraints in Optimization* on page 411. As that chapter explains, logical constraints in ILOG CPLEX are available only to C++ users.

IloCplex and MP Models

What is special about that list of Concert Technology classes recognized by an instance of `IloCplex`? A model consisting of instances of those classes can be transformed into a MIP or LP in the conventional form:

Maximize (or minimize) an objective function

such that `LowerBounds <= Ax <= UpperBounds`

for every `lowerBound <= x <= upperBound`

When all the variables, indicated by `x`, are continuous floating-point variables, a problem in this conventional form is known as a linear programming model (LP). When some variables are integer- or Boolean-valued, a problem in this form is known as a mixed integer programming model (MIP).

At first glance, it might appear that this formulation greatly restricts the kinds of problems that can be modeled in this way. However, practice has shown that an astonishing variety of problems can be represented by such a model. The book *Model Building in Mathematical Programming* by H.P. Williams offers a good starting point if you are interested in model-building techniques for LPs and MIPs.

IloCplex and Algorithms

An important observation to make here is that an instance of `IloCplex` is not really *one* algorithm, though it is referred to as an algorithm since `IloCplex` is a subclass of `IloAlgorithm`. In fact, an instance of `IloCplex` consists of a set of highly configurable algorithms (also known as optimizer options). They include primal and dual simplex

algorithms, barrier algorithms, a network simplex solver and a branch & cut solver for MIPs. In most cases, you can use `IloCplex` like a "black box" without concern about the options, though you can also select the optimizer options individually yourself. The options provide a wealth of parameters that allow you to fine-tune the algorithm to your particular model. Other parts of the *ILOG CPLEX User's Manual* contain detailed instructions about exploiting these options.

In the case of MIPs, for example, you can use your own callback directly to control the branch & cut search carried out by `IloCplex`.

In pure LPs (that is, linear programs with no integer variables, no Boolean variables, no semi-continuous variables, no piecewise linear functions, no logical constraints, no special ordered sets (SOSs)), you can query `IloCplex` for additional information, such as dual information or basis information (with appropriate options). `IloCplex` also supports sensitivity analysis indicating how to modify your model while preserving the same solution. In cases where your "solution" is infeasible, the infeasibility finder enables you to analyze the source of infeasibility.

What Are Semi-Continuous Variables?

A semi-continuous variable is a variable that by default can take the value 0 (zero) or any value between its semi-continuous lower bound (sclb) and its upper bound (ub). Both those bounds (the sclb and the ub) must be finite. In Concert Technology, semi-continuous variables are represented by the class `IloSemiContVar`. To create a semi-continuous variable, you use the constructor from that class to indicate the environment, the semi-continuous lower bound, and the upper bound of the variable, like this:

```
IloSemiContVar mySCV(env, 1.0, 3.0);
```

That statement creates a semi-continuous variable with a semi-continuous lower bound of 1.0 and an upper bound of 3.0. The method `IloSemiContVar::getSemiContinuousLB` returns the semi-continuous lower bound of the invoking variable, and the method `IloSemiContVar::getUB` returns the upper bound. That class, its constructors, and its methods are documented in the *Concert Technology Reference Manual*.

In that manual, you will see that `IloSemiContVar` derives from `IloNumVar`, the Concert Technology class for numeric variables. Like other numeric variables, semi-continuous variables assume floating-point values by default (type `ILOFLOAT`). However, you can designate a semi-continuous variable as integer (type `ILOINT`). In that case, it is a *semi-integer* variable.

For details about the feasible region of a semi-continuous or semi-integer variable, see the documentation of `IloSemiContVar` in the *Concert Technology Reference Manual*.

Describing the Problem

With this background about semi-continuous variables, consider an example using them. Assume that you are managing a power plant of several generators. Each of the generators may be on or off (producing or not producing power). When a generator is on, it produces power between its minimum and maximum level, and each generator has its own minimum and maximum levels. The cost for producing a unit of output differs for each generator as well. The aim of the problem is to satisfy demand for power while minimizing cost in the best way possible.

Representing the Problem

As input for this example, you need such data as the minimum and maximum output level for each generator. The application will use Concert Technology arrays `minArray` and `maxArray` for that data. It will read data from a file into these arrays, and then learn their length (that is, the number of generators available) by calling the method `getSize`.

The application also needs to know the cost per unit of output for each generator. Again, a Concert Technology array, `cost`, serves that purpose as the application reads data in from a file with the `operator >>`.

The application also needs to know the demand for power, represented as a numeric variable, `demand`.

Building a Model

After the application creates an environment and a model in that environment, it is ready to populate the model with extractable objects pertinent to the problem.

It represents the production level of each generator as a semi-continuous variable. In that way, with the value 0 (zero), the application can accommodate whether the generator is on or off; with the semi-continuous lower bound of each variable, it can indicate the minimum level of output from each generator; and indicate the maximum level of output for each generator by the upper bound of its semi-continuous variable. The following lines create the array `production` of semi-continuous variables (one for each generator), like this:

```
IloNumVarArray production(env);
   for (IloInt j = 0; j < generators; ++j)
        production.add(IloSemiContVar(env, minArray[j], maxArray[j]));
```

The application adds an objective to the model to minimize production costs in this way:

```
mdl.add(IloMinimize(env, IloScalProd(cost, production)));
```

It also adds a constraint to the model: it must meet demand.

```
mdl.add(IloSum(production) >= demand);
```

With that model, now the application is ready to create an algorithm (in this case, an instance of `IloCplex`) and extract the model.

Solving the Problem

To solve the problem, create the algorithm, extract the model, and solve.

Ending the Program

As in all C++ CPLEX applications, this program ends with a call to `IloEnv::end` to de-allocate the model and algorithm once they are no longer in use.

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

Complete Program

You can see the entire program here or on-line in the standard distribution of ILOG CPLEX in the file `/examples/src/rates.cpp`. To run that example, you need a license for ILOG CPLEX.

```
// File: examples/src/rates.cpp
// Version 9.0
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// rates.cpp -- modeling with semi-continuous variables
//
// Problem Description:
//
// Assume you run a power supply company.  You have several power generators
// available, each of which has a minimum and maximum production level
// and a cost per unit output.
// The question is which generators to use in order to
// minimize the overall operation cost while satisfying the demand.
//
```

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

int
main(int argc, char** argv)
{
   IloEnv env;
   try {
      IloNumArray minArray(env), maxArray(env), cost(env);
      IloNum demand;
      if ( argc < 2 ) {
         env.warning()
           << "Default data file : ../../../examples/data/rates.dat" << endl;
         ifstream in("../../../examples/data/rates.dat");
         in >> minArray >> maxArray >> cost >> demand;
      } else {
         ifstream in(argv[1]);
         in >> minArray >> maxArray >> cost >> demand;
      }

      IloModel mdl(env);

      IloNumVarArray production(env);
      IloInt generators = minArray.getSize();
      for (IloInt j = 0; j < generators; ++j) {
         production.add(IloSemiContVar(env, minArray[j], maxArray[j]));
      }

      mdl.add(IloMinimize(env, IloScalProd(cost, production)));
      mdl.add(IloSum(production) >= demand);

      IloCplex cplex(mdl);
      cplex.exportModel("rates.lp");
      if (cplex.solve()) {
         for (IloInt j = 0; j < generators; ++j) {
            cplex.out() << "   generator " << j << ": "
                        << cplex.getValue(production[j]) << endl;
         }
         cplex.out() << "Total cost = " << cplex.getObjValue() << endl;
      }
      else cplex.out()<< "No solution" << endl;
      cplex.printTime();
   }
   catch (IloException& ex) {
      cerr << "Error: " << ex << endl;
   }
   catch (...) {
      cerr << "Error" << endl;
   }
   env.end();
   return 0;
```

```
}
```

16

Transport: Piecewise Linear Optimization

This chapter shows you how to represent piecewise linear optimization in ILOG CPLEX with Concert Technology. In this chapter, you will learn:

- how to use classes of Concert Technology to represent piecewise linear functions;
- how to determine whether branching is necessary in a piecewise linear problem.

Piecewise Linearity in ILOG CPLEX

Some problems are most naturally represented by constraints over functions that are not purely linear but consist of linear *segments*. Such functions are sometimes known as *piecewise linear*. In this chapter, a transportation example shows you various ways of stating and solving problems that lend themselves to a piecewise linear model. Before plunging into the problem itself, this section defines a few terms appearing in this discussion.

What Is a Piecewise Linear Function?

From a geometric point of view, Figure 16.1 shows a conventional piecewise linear function `f(x)`. This particular function consists of four segments. If you consider the function over four separate intervals, `(-∞, 4)` and `[4, 5)` and `[5, 7)` and `[7, ∞)`, you see that `f(x)` is linear in each of those separate intervals. For that reason, it is said to be *piecewise linear*. Within each of those segments, the slope of the linear function is clearly increasing or

decreasing. The points where the slope of the function changes are known as *breakpoints*. The piecewise linear function in Figure 16.1 has three breakpoints.

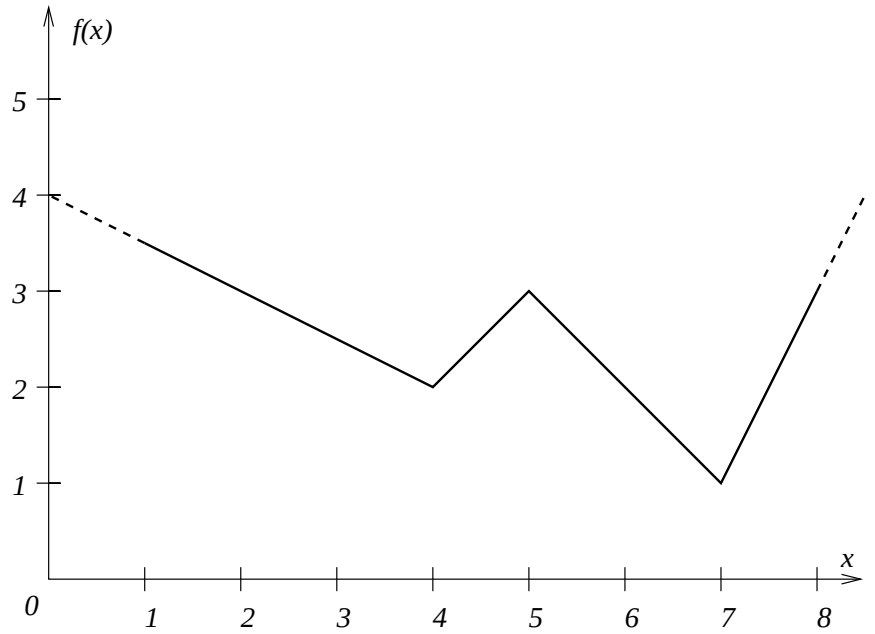

Figure 16.1 *A piecewise linear function with breakpoints*

Piecewise linear functions are often used to represent or to approximate nonlinear unary functions (that is, nonlinear functions of one variable). For example, piecewise linear functions frequently represent situations where costs vary with respect to quantity or gains vary over time.

Syntax of Piecewise Linear Functions

To define a piecewise linear function in Concert Technology, you need these components:

- the variable of the piecewise linear function;
- the breakpoints of the piecewise linear function;
- the slope of each segment (that is, the rate of increase or decrease of the function between two breakpoints);
- the geometric coordinates of at least one point of the function.

In other words, for a piecewise linear function of `n` breakpoints, you need to know `n+1` slopes.

Typically, the breakpoints of a piecewise linear function are specified as an array of numeric values. For example, the breakpoints of the function `f(x)` as it appears in Figure 16.1 are specified in this way:

```
IloNumArray (env, 3, 4., 5., 7)
```

The slopes of its segments are indicated as an array of numeric values as well. For example, the slopes of `f(x)` are specified in this way:

```
IloNumArray (env, 4, -0.5, 1., -1., 2.)
```

The geometric coordinates of at least one point of the function, `(x, f(x))` must also be specified; for example, `(4, 2)`. Then in Concert Technology, those elements are brought together in an instance of the class `IloPiecewiseLinear` in this way:

```
IloPiecewiseLinear(x,
                   IloNumArray(env, 3, 4., 5., 7.),
                   IloNumArray(env, 4, -0.5, 1., -1., 2.),
                   4, 2)
```

Another way to specify a piecewise linear function is to give the slope of the first segment, two arrays for the coordinates of the breakpoints, and the slope of the last segment. In this approach, the example `f(x)` from Figure 16.1 looks like this:

```
IloPiecewiseLinear(x, -0.5, IloNumArray(env, 3, 4., 5., 7.),
                            IloNumArray(env, 3, 2., 3., 1.), 2);
```

Discontinuous Piecewise Linear Functions

Thus far, you have seen a piecewise linear function where the segments are *continuous*. Intuitively, in a continuous piecewise linear function, the endpoint of one segment has the same coordinates as the initial point of the next segment, as in Figure 16.1.

There are piecewise linear functions, however, where two consecutive breakpoints may have the same `x` coordinate but differ in the value of `f(x)`. Such a difference is known as a *step* in the piecewise linear function, and such a function is known as *discontinuous*. Figure 16.2 shows a discontinuous piecewise linear function with two steps.

Syntactically, a step is represented in this way:

- The value of the first point of a step in the array of slopes is the *height* of the step.
- The value of the second point of the step in the array of slopes is the *slope* of the function after the step.

By convention, a breakpoint belongs to the segment that starts at that breakpoint. For example, in Figure 16.2, `f(3)` is equal to `3` (not `1`); similarly, `f(5)` is equal to `5`.

In Concert Technology, a discontinuous piecewise linear function is represented as an instance of the class `IloPiecewiseLinear` (the same class as used for continuous piecewise linear functions). For example, the function in Figure 16.2 is declared in this way:

```
IloPiecewiseLinear(x,
                   IloNumArray(env, 4, 3. ,3. ,5. ,5.),
                   IloNumArray(env, 5, 0., 2., 0.5, 1., -1.),
                   0, 1);
```

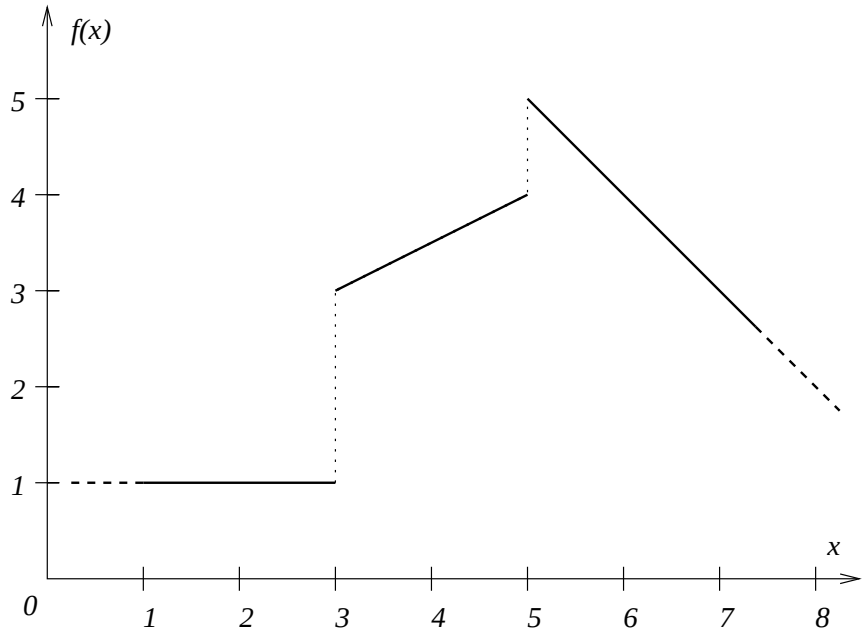

Figure 16.2 *A discontinuous piecewise linear function with steps*

Using IloPiecewiseLinear

Whether it represents a continuous or a discontinuous piecewise linear function, an instance of `IloPiecewiseLinear` behaves like a floating-point *expression*. That is, you may use it in a term of a linear expression or in a constraint added to a model (an instance of `IloModel`).

Special Considerations about Solving with IloPiecewiseLinear

Under some conditions, a piecewise linear problem may be equivalent to a linear program (rather than a MIP). In such a case, no branching is necessary. Whether a problem represented by a piecewise linear function made up of segments can be modeled as a linear

program (and thus it can be solved straightforwardly without branching) depends on the slope of these segments.

A piecewise linear function is *convex* if the succession of the slopes of its segments is increasing. In contrast, a piecewise linear function is *concave* if the succession of slopes is decreasing.

Consider a minimization problem having linear constraints of the form

$a_1x_1+...+a_nx_n <= a_0$ and

$a_1x_1+...+a_nx_n = a_0$ where x_i is a variable or a piecewise linear function.

Sufficient conditions for this problem to be a linear program are these:

- every piecewise linear function is convex or concave;
- every convex function appears with a positive coefficient in <= constraints or in the objective function;
- every concave function appears with a negative coefficient in <= constraints or in the objective function.

These statements have equivalent translations for >= constraints or for a maximization problem. See *Are Logical Constraints Discrete or Continuous?* on page 416 for more detail about piecewise linear functions in the context of whether they are convex or concave and for an indication of when piecewise linear functions lead to a MIP or LP formulation.

With this background about piecewise linear functions and their representation in Concert Technology, consider a transportation example exploiting piecewise linearity.

Describing the Problem

Assume that a company must ship cars from factories to showrooms. Each factory can supply a fixed number of cars, and each showroom needs a fixed number of cars. There is a cost for shipping a car from a given factory to a given showroom. The objective is to minimize the total shipping cost while satisfying the demands and respecting supply.

In concrete terms, assume there are three factories and four showrooms. Here is the quantity that each factory can supply:

```
supply0 = 1000
supply1 = 850
supply2 = 1250
```

Each showroom has a fixed demand:

$demand_0 = 900$
$demand_1 = 1200$
$demand2 = 600$
$demand3 = 400$

Let `nbSupply` be the number of factories and `nbDemand` be the number of showrooms. Let x_{ij} be the number of cars shipped from factory `i` to showroom `j`. The model is composed of `nbDemand` + `nbSupply` constraints that force all demands to be satisfied and all supplies to be shipped. Thus far, a model for our problem looks like this:

Minimize $$\sum_{i=0}^{nbDemand-1} \sum_{j=0}^{nbSupply-1} cost_{ij} \cdot x_{ij}$$

subject to

$$\sum_{j=0}^{nbSupply-1} x_{ij} = supply_i \qquad i = 0, \ldots, nbDemand-1$$

$$\sum_{i=0}^{nbDemand-1} x_{ij} = demand_j \qquad j = 0, \ldots, nbSupply-1$$

Variable Shipping Costs

Now consider the costs of shipping from a given factory to a given showroom. Assume that for every pair (factory, showroom), there are different rates, varying according to the quantity shipped. To illustrate the difference between convex and concave piecewise linear functions, in fact, this example assumes that there are two different tables of rates for shipping cars from factories to showrooms. The first table of rates looks like this:

- a rate of 120 per car for quantities between 0 and 200;
- a rate of 80 per car for quantities between 200 and 400;
- a rate of 50 per car for quantities higher than 400.

These costs that vary according to quantity define the piecewise linear function represented in Figure 16.3. As you see, the slopes of the segments of that function are decreasing, so that function is concave.

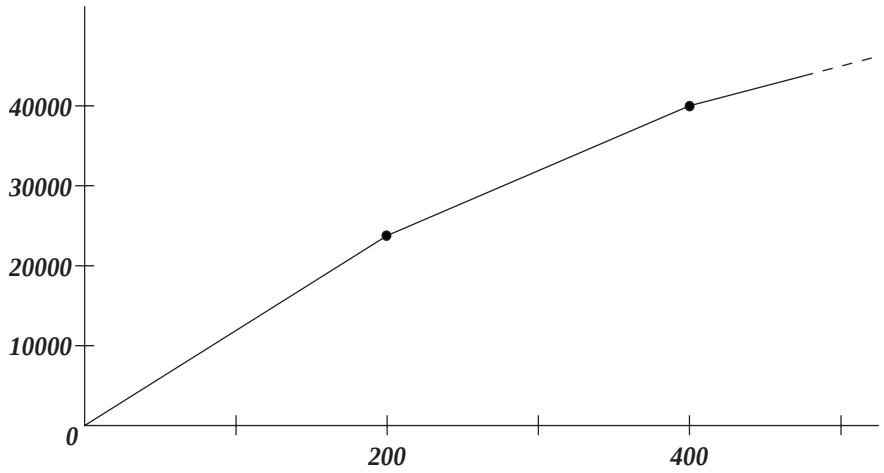

Figure 16.3 *A concave piecewise linear cost function*

Also assume that there is a second table of rates for shipping cars from factories to showrooms. The second table of rates looks like this:

- a rate of 30 per car for quantities between 0 and 200;
- a rate of 80 per car for quantities between 200 and 400;
- a rate of 130 per car for quantities higher than 400.

The costs in this second table of rates that vary according to the quantity of cars shipped define a piecewise linear function, too. It appears in Figure 16.4. The slopes of the segments in this second piecewise linear function are increasing, so this function is convex.

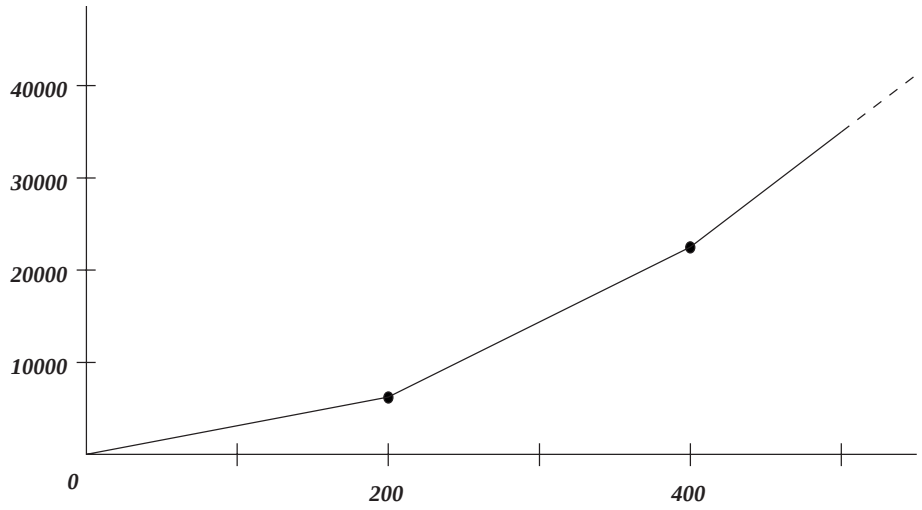

Figure 16.4 *A convex piecewise linear cost function*

Model with Varying Costs

With this additional consideration about costs varying according to quantity, our model now looks like this:

minimize $$\sum_{i=0}^{nbDemand-1} \sum_{j=0}^{nbSupply-1} y_{ij}$$

subject to

$y_{ij} = f(x_{ij})$ for $i = 0, ..., nbDemand-1$ and $j = 0, ..., nbSupply-1$

$$\sum_{j=0}^{nbSupply-1} x_{ij} = demand_i \qquad \text{for} \qquad i = 0, \ldots, nbDemand-1$$

$$\sum_{i=0}^{nbDemand-1} x_{ij} = supply_j \qquad \text{for} \qquad j = 0, \ldots, nbSupply-1$$

With this problem in mind, consider how to represent the data and model in Concert Technology.

Representing the Data

As in other examples, the template class `IloArray` appears in a type definition to create matrices for this problem, like this:

```
typedef IloArray<IloNumArray>    NumMatrix;
typedef IloArray<IloNumVarArray> NumVarMatrix;
```

Those two-dimensional arrays (that is, arrays of arrays) are now available in the application to represent the demands from the showrooms and the supplies available from the factories.

```
IloInt nbDemand = 4;
IloInt nbSupply = 3;
IloNumArray supply(env, nbSupply, 1000., 850., 1250.);
IloNumArray demand(env, nbDemand, 900., 1200., 600., 400.);

NumVarMatrix x(env, nbSupply);
NumVarMatrix y(env, nbSupply);
```

Developing a Model

As in other examples in this manual, this application begins by creating an environment, an instance of `IloEnv`, documented in the *Concert Technology Reference Manual*.

```
IloEnv env;
```

Within that environment, a model for this problem is created as an instance of `IloModel`, also documented in the *Concert Technology Reference Manual.*

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

Then constraints and an objective are added to the model.

Adding Constraints

According to the description of the problem, the supply of cars from the factories must meet the demand of the showrooms. At the same time, it is important not to ship cars that are not

in demand; in terms of this model, the demand should meet the supply as well. Those ideas are represented as constraints added to the model, like this:

```
for(i = 0; i < nbSupply; i++){
  x[i] = IloNumVarArray(env, nbDemand, 0, IloInfinity, ILOFLOAT);
  y[i] = IloNumVarArray(env, nbDemand, 0, IloInfinity, ILOFLOAT);
}

for(i = 0; i < nbSupply; i++){      // supply must meet demand
  model.add(IloSum(x[i]) == supply[i]);
}
for(j = 0; j < nbDemand; j++){      // demand must meet supply
  IloExpr v(env);
  v.end();
}
```

Checking Convexity and Concavity

To illustrate the ideas of convex and concave piecewise linear functions, two tables of costs that vary according to the quantity of cars shipped were introduced in the problem description. To accommodate those two tables in the model, the following lines are added.

```
if (convex) {
  for(i = 0; i < nbSupply; i++){
    for(j = 0; j < nbDemand; j++){
      model.add(y[i][j] == IloPiecewiseLinear(x[i][j],
                                    IloNumArray(env, 2, 200., 400.),
                                    IloNumArray(env, 3, 30., 80., 130.),
                                    0, 0));
    }
  }
}else{
  for(i = 0; i < nbSupply; i++){
    for(j = 0; j < nbDemand; j++){
      model.add(y[i][j] == IloPiecewiseLinear(x[i][j],
                                    IloNumArray(env, 2, 200., 400.),
                                    IloNumArray(env, 3, 120., 80., 50.),
                                    0, 0));
    }
  }
}
```

Adding an Objective

The objective is to minimize costs of supplying cars from factories to showrooms, It is added to the model in these lines:

```
IloExpr obj(env);
for(i = 0; i < nbSupply; i++){
  obj += IloSum(y[i]);
}

model.add(IloMinimize(env, obj));
```

Solving the Problem

If the problem satisfies the special conditions of a simple linear program, as outlined in *Special Considerations about Solving with IloPiecewiseLinear* on page 400, then an instance of `IloCplex` (documented in the *ILOG CPLEX Reference Manual*) will solve the problem with a simplex linear optimizer.

On the other hand, if the problem fits the more general case, and thus requires a mixed integer programming (MIP) for its solution, an instance of `IloCplex` will solve the problem with a MIP optimizer using branch & bound techniques.

In either case (LP or MIP), the application needs only the following few lines. They create an algorithm (an instance of `IloCplex` documented in the *ILOG CPLEX Reference Manual*) in an environment (an instance of `IloEnv` documented in the *ILOG Concert Reference Manual*) and extract the model (an instance of `IloModel` also documented in the *Concert Reference Manual*) for that algorithm to find a solution.

```
IloCplex cplex(env);
cplex.extract(model);
cplex.solve();
```

Displaying a Solution

To display the solution, use the methods of `IloEnv` (documented in the *ILOG Concert Reference Manual*) and `IloCplex` (documented in the *ILOG CPLEX Reference Manual*).

```
env.out() << " - Solution: " << endl;
for(i = 0; i < nbSupply; i++){
  env.out() << "   " << i << ": ";
  for(j = 0; j < nbDemand; j++){
    env.out() << cplex.getValue(x[i][j]) << "\t";
  }
  env.out() << endl;
}
env.out() << "   Cost = " << cplex.getObjValue() << endl;
```

Ending the Application

As in other C++ examples in this manual, the application ends with a call to the method `IloEnv::end` to clean up the memory allocated for the environment and algorithm.

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

Complete Program: transport.cpp

You can see the complete program here or on line in the standard distribution of `IloCplex` in the file `/examples/src/transport.cpp`. To run this example, you need a license for ILOG CPLEX.

```
// File: examples/src/transport.cpp
// Version 9.0
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//

#include <ilcplex/ilocplex.h>

ILOSTLBEGIN

typedef IloArray<IloNumArray>    NumMatrix;
typedef IloArray<IloNumVarArray> NumVarMatrix;

int main(int argc, char** argv) {
   if (argc <= 1) {
      cerr << "Usage: " << argv[0] << " <model>" << endl;
      cerr << "  model = 0 -> convex piecewise linear model, " << endl;
      cerr << "  model = 1 -> concave piecewise linear model. [default]" <<
endl;
   }

   IloBool convex;
   if (argc <= 1)
      convex = IloFalse;
   else
      convex = atoi(argv[1]) == 0 ? IloTrue : IloFalse;

   IloEnv env;
   try {
      IloInt i, j;
      IloModel model(env);

      IloInt nbDemand = 4;
      IloInt nbSupply = 3;
      IloNumArray supply(env, nbSupply, 1000., 850., 1250.);
      IloNumArray demand(env, nbDemand, 900., 1200., 600., 400.);

      NumVarMatrix x(env, nbSupply);
      NumVarMatrix y(env, nbSupply);

      for(i = 0; i < nbSupply; i++) {
         x[i] = IloNumVarArray(env, nbDemand, 0, IloInfinity, ILOFLOAT);
```

```
      y[i] = IloNumVarArray(env, nbDemand, 0, IloInfinity, ILOFLOAT);
   }

   for(i = 0; i < nbSupply; i++) {      // supply must meet demand
      model.add(IloSum(x[i]) == supply[i]);
   }
   for(j = 0; j < nbDemand; j++) {      // demand must meet supply
      IloExpr v(env);
      for(i = 0; i < nbSupply; i++)
         v += x[i][j];
      model.add(v == demand[j]);
      v.end();
   }
   if (convex) {
      for(i = 0; i < nbSupply; i++) {
         for(j = 0; j < nbDemand; j++) {
            model.add(y[i][j] == IloPiecewiseLinear(x[i][j],
                                    IloNumArray(env, 2, 200., 400.),
                                    IloNumArray(env, 3, 30., 80., 130.),
                                    0, 0)
);
         }
      }
   }
   else {
      for(i = 0; i < nbSupply; i++) {
         for(j = 0; j < nbDemand; j++) {
            model.add(y[i][j] == IloPiecewiseLinear(x[i][j],
                                    IloNumArray(env, 2, 200., 400.),
                                    IloNumArray(env, 3, 120., 80., 50.),
                                    0, 0)
);
         }
      }
   }

   IloExpr obj(env);
   for(i = 0; i < nbSupply; i++) {
      obj += IloSum(y[i]);
   }
   model.add(IloMinimize(env, obj));

   IloCplex cplex(env);
   cplex.extract(model);
   cplex.exportModel("transport.lp");
   cplex.solve();

   env.out() << " - Solution: " << endl;
   for(i = 0; i < nbSupply; i++) {
      env.out() << "   " << i << ": ";
      for(j = 0; j < nbDemand; j++) {
         env.out() << cplex.getValue(x[i][j]) << "\t";
      }
```

```
            env.out() << endl;
         }
         env.out() << "   Cost = " << cplex.getObjValue() << endl;
      } catch(IloException& e) {
         cerr << "ERROR: " << e.getMessage() << endl;
      } catch (...) {
         cerr << "Error" << endl;
      }
      env.end();
      return 0;
   } // END main
```

17

Logical Constraints in Optimization

This chapter shows you how to represent logical constraints in ILOG CPLEX with Concert Technology for C++ users. In C++ applications, Concert Technology can automatically translate logical constraints into their transformed equivalent that the discrete (that is, MIP) or continuous optimizers of ILOG CPLEX can process efficiently.

In this chapter, you will learn:

- *What Are Logical Constraints?* on page 412
- *What Can CPLEX Extract from a Model with Logical Constraints?* on page 412
- *Which Logical Expressions Can CPLEX Extract?* on page 413
- *What Does CPLEX Transform?* on page 414
- *Are Logical Constraints Discrete or Continuous?* on page 416

> ***Note:*** *These logical constraint facilities are available in ILOG CPLEX only with Concert Technology for C++ users.*

What Are Logical Constraints?

For ILOG CPLEX, a *logical constraint* combines linear constraints by means of logical operators, such as logical-and, logical-or, negation (that is, not), conditional statements (that is, if ... then ...) to express complex relations between linear constraints. ILOG CPLEX can also handle certain logical expressions appearing within a linear constraint. One such logical expression is the minimum of a set of variables. Another such logical expression is the absolute value of a variable. There's more about logical expressions in *Which Logical Expressions Can CPLEX Extract?* on page 413.

What Can CPLEX Extract from a Model with Logical Constraints?

In C++ applications, the class `IloCplex` can extract modeling objects to solve a wide variety of MIPs, as you see in *Solving Concert Technology Models with IloCplex* on page 47, summarized in Table 1.1 on page 49. In fact, the class `IloCplex` can extract logical constraints as well as some logical expressions. The logical constraints that `IloCplex` can extract are these:

- `IloAnd`
- `IloOr`
- `IloNot`
- `IloIfThen`
- == that is, the equivalence relation
- `!=` that is, the exclusive-or relation

Among those extractable objects, `IloAnd` `IloOr`, and `IloNot` can also be represented in your application by means of the C++ operators:

- `||` (for `IloOr`)
- && (for `IloAnd`)
- `!` (for `IloNot`)

All those extractable objects accept as their arguments other linear constraints or logical constraints, so you can combine linear constraints with logical constraints in complicated expressions in your application.

For example, to express the idea that two jobs with starting times `x1` and `x2` and with duration `d1` and `d2` must not overlap, you can either use overloaded C++ operators, like this:

```
model.add((x1 >= x2 + d2) || (x2 >= x1 + d1));
```

or you can express the same idea, like this:

```
IloOr or(env)
or.add(x1 >= x2 + d2);
or.add(x2 >= x1 + d1);
model.add(or);
```

Since `IloCplex` can also extract logical constraints embedded in other logical constraints, you can also write logical constraints like this:

```
IloIfThen(env, (x >= y && x >= z), IloNot(x <= 300 || y >= 700))
```

where `x`, `y`, and `z` are variables in your application.

Which Logical Expressions Can CPLEX Extract?

Some expressions are easily recognized as *nonlinear*, for example, a function such as $x^2 + y^2 \leq 1$. However, other nonlinearities are less obvious, such as absolute value as a function. In a very real sense, MIP is a class of nonlinearly constrained problems because the integrality restriction destroys the property of convexity which the constraints otherwise might possess. Because of that characteristic, certain (although not all) nonlinearities are capable of being converted to a MIP formulation, and thus can be solved by ILOG CPLEX. In fact, `IloCplex` can extract the following nonlinear expressions in a C++ application:

- `IloMin` the minimum of an array of numerical expressions or over a numerical expression and a constant
- `IloMax` the maximum of an array of numerical expressions or over a numerical expression and a constant
- `IloAbs` the absolute value of a numerical expression
- `IloPiecewiseLinear` the piecewise linear combination of a numerical expression,
- A linear constraint can appear as a term in a logical constraint.

For example, given these variables and arrays:

```
IloIntVarArray x(env, 5, 0, 1000);
IloNumVar y(env, -1000, 5000);
IloNumVar z(env, -1000, 1000);
```

`IloCplex` recognizes the following constraint as valid and extracts it:

```
IloMin(x) >= IloAbs(y)
```

In fact, ranges containing logical expressions can, in turn, appear in logical constraints. For example, the following constraint is valid and extractable by `IloCplex`:

```
IloIfThen(env, (IloAbs(y) <= 100), (z <= 300));
```

Linear constraints can appear as terms in logical constraints. A linear constraint appearing as a term in a logical constraint behaves like a binary value. For example, given x and y as variables, you can write the following lines to get the truth value of $x \geq y$ in a binary value:

```
IloIntVar b(env, 0, 1);
model.add(b == (x >= y));
```

> ***Reminder:*** *In mathematics, the numbers used for counting are known as the natural numbers or as cardinal numbers. Cardinality thus means "how many?" as in how many elements there are in a set. ILOG CPLEX offers efficient ways to represent constraints for counting.*

Optimizing how many items appear in a solution is often an issue in practical problems. Questions of counting (how many?) can be represented formally as *cardinality* constraints. For example, suppose that your application includes three variables, each representing a quantity of one of three products, and assume further that a good solution to the problem means that the quantity of at least two of the three products must be greater than 20. Then you can represent that idea in your application, like this:

```
IloNumVarArray x(env, 3, 0, 1000);
model.add((x[0] >= 20) + (x[1] >= 20) + (x[2] >= 20) >= 2);
```

It is important to note here that only linear constraints can appear as arguments of logical constraints extracted by `IloCplex`. That is, quadratic constraints are not handled in logical constraints. Similarly, quadratic terms cannot appear as arguments of logical expressions such as `IloMin`, `IloMax`, `IloAbs`, and `IloPiecewiseLinear`.

What Does CPLEX Transform?

In ILOG CPLEX with ILOG Concert Technology for C++ users, logical constraints and logical expressions are *transformed at extraction*, meaning that `IloCplex` transforms them into an equivalent set of constraints that may introduce additional discrete variables to the formulation along with integrality restrictions.

For example, look again at the constraint that you saw in the scheduling problem of two jobs with durations that must not overlap, and consider it in the context of an application that exports the model to a file in LP format, like this:

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

IloNumVar x1(env, 0, 2000, "x1");
IloNumVar x2(env, 0, 2000, "x2");
model.add((x1 >= x2 + 100) || (x2 >= x1 + 30));

IloCplex cplex(model);
cplex.exportModel("logic_example.lp");
env.end();
```

After you execute that application, the file `logic_example.lp` contains this model:

```
Minimize
 obj:
Subject To
 id10: x1 - x2 - 2100 b >= -2000
 id12: - x1 + x2 + 2030 b >= 30
Bounds
 0 <= x1 <= 2000
 0 <= x2 <= 2000
 0 <= b <= 1
Generals
 b
End
```

You can see in that file that a binary variable `b` has been introduced to express the logical constraint, and two linear constraints have been added. When `b = 0`, the constraint `x2 >= x1 + 30` is satisfied, thanks to the linear constraint `id12`. If `b = 1`, then the constraint `x1 >= x2 + 100` is satisfied, thanks to the linear constraint `id10`.

ILOG CPLEX uses this kind of transformation for every logical constraint and every logical term.

To get an idea of how cardinality (counting how many) is transformed, look again at the cardinality constraint in the problem of three products of differing quantities, where a good solution entails quantities greater than 20 for at least two of the three products.

```
IloNumVarArray x(env, 3, 0, 1000);
model.add((x[0] >= 20) + (x[1] >= 20) + (x[2] >= 20) >= 2);
```

The exported model of that constraint in LP format looks like this:

```
Problem name: card.lp

Minimize
 obj:
Subject To
 id15: x1 - 20 id13 >= 0
 id18: x2 - 20 id16 >= 0
 id21: x3 - 20 id19 >= 0
 id22: id13 + id16 + id19 >= 2
Bounds
 0 <= x1 <= 1000
 0 <= x2 <= 1000
 0 <= x3 <= 1000
 0 <= id13 <= 1
 0 <= id16 <= 1
 0 <= id19 <= 1
Generals
 id13  id16  id19
End
```

That model shows how ILOG CPLEX transforms logical terms: it introduces three binary variables, one for each constraint. When the value of a binary variable is 1, then the corresponding constraint `x[i] >= 20` must be satisfied as well.

Are Logical Constraints Discrete or Continuous?

Under some conditions, a problem expressed in terms of logical constraints may be equivalent to a continuous problem, rather than a discrete problem (that is, a MIP). In such a case, there is no need for branching during the search for a solution. Whether a problem (or parts of a problem) represented by logical terms can be modeled and solved by a continous optimizer depends on the shape of those logical terms. In this context, shape means convex or concave in the formal, mathematical sense. (For an intuitive idea of convex and concave in a different context, see also *Convexity* on page 296.)

For example, absolute value, as represented in Concert Technology by `IloAbs(x)`, is convex, whereas `-IloAbs(x)` is concave.

Similarly, a piecewise linear function is convex if the succession of the slopes of its segments is increasing over the interval of the term it is applied to. In contrast, a piecewise linear function is concave if the succession of slopes is decreasing. (For more detail about convexity and piecewise linearity, see also *Special Considerations about Solving with IloPiecewiseLinear* on page 400.)

Consequently, a logical term has a continuous formulation in ILOG CPLEX if it satisfies one of the following conditions:

- It is convex and every occurrence of this term appears with a positive or zero coefficient in less-than-or-equal-to constraints (<=), with a negative or zero coefficient in greater-than-or-equal-to constraints (>=), and with a positive or zero coefficient in a minimization objective function.
- It is concave and every occurrence of this term appears with a negative or zero coefficient in less-than-or-equal-to constraints (<=), with a positive or zero coefficient in greater-than-or-equal-to constraints (>=), and with a negative or zero coefficient in a maximization objective function.

As an example, consider the following logical term in a minimization:

```
IloNumVarArray x(env, 3, 0, 10));
model.add(IloMinimize(env, IloMax(x)));
```

In that example, `IloMax` is convex. Since it appears in a minimization objective function, it has a linear programming formulation. In fact, if you export the model of this example in LP format, it looks like this:

```
Minimize
 obj: y
Subject To
 id9:  - x1 + y >= 0
 id11: - x2 + y >= 0
 id13: - x3 + y >= 0
Bounds
 0 <= x1 <= 10
 0 <= x2 <= 10
 0 <= x3 <= 10
 0 <= y <= 10
End
```

As you can see in the exported model, a new variable, `y`, has been created. The problem is to minimize `y` while requiring `y` to be greater than each of the other variables in the array `x`.

In contrast, if you replace the minimization objective function by a maximization, then a MIP formulation becomes necessary, as you can see when the modified model is exported to a file.

```
Maximize
 obj: y
Subject To
 id12: - x1 + y >= 0
 id14: - x1 + y + 10 id8 <= 10
 id16: - x2 + y >= 0
 id18: - x2 + y + 10 id9 <= 10
 id20: - x3 + y >= 0
 id22: - x3 + y + 10 id10 <= 10
 id24: id8 + id9 + id10  = 1
Bounds
 0 <= x1 <= 10
 0 <= x2 <= 10
 0 <= x3 <= 10
 0 <= y <= 10
 0 <= id8 <= 1
 0 <= id9 <= 1
 0 <= id10 <= 1
Generals
 id8  id9  id10
End
```

The MIP formulation contains all the constraints from the continuous formulation, but now `y` must be maximized and in addition must be equal to exactly one of the `x[i]`. For this purpose, three binary variables have been introduced (`id8`, `id9`, and `id10`). When one of these three is set to 1, the corresponding variable `x[i]` must also satisfy the constraint `y <= x[i]`, in addition to those in the continuous formulation.

In short, it may be difficult to forecast accurately whether a model containing logical expressions will be extracted as continuous or discrete. For example, a small change in the formulation or a particular instance of the input data may destroy the property of convexity that previously had been present.

In most applications, it will be safest to assume the extracted model always will be a MIP, because in cases where it is actually continuous, the optimizer will treat it correctly anyway, namely, as the degenerate case having no integrality restrictions to apply.

A particular caution: it may be unsafe to assume that post-solution information involving dual values corresponding to a continuous model (for example, values from the method `getDual`) will be obtainable or will have any useful meaning in a model containing logical constraints.

C H A P T E R

18

Food Manufacturing 2: Using Logical Constraints

Chapter 17, *Logical Constraints in Optimization*, introduced features of ILOG CPLEX that transform parts of your problem automatically for you. This chapter shows you some of those features in use. The chapter is based on the formulation by H.P. Williams of a standard industrial problem in food manufacturing. The aim of the problem is to blend a number of oils cost effectively in monthly batches. In this version of the problem, formulated by Williams as food manufacturing 2 in his book *Model Building in Mathematical Programming*, the number of ingredients in a blend must be limited, and extra conditions are added to govern which oils can be blended.

> ***Note:*** *These logical constraint facilities are available in ILOG CPLEX only with Concert Technology for C++ users.*

Describing the Problem

The problem is to plan the blending of five kinds of oil, organized in two categories (two kinds of vegetable oils and three kinds of non vegetable oils) into batches of blended products over six months.

Some of the oil is already available in storage. There is an initial stock of oil of 500 tons of each raw type when planning begins. An equal stock should exist in storage at the end of the plan. Up to 1000 tons of each type of raw oil can be stored each month for later use. The price for storage of raw oils is 5 monetary units per ton. Refined oil cannot be stored. The blended product cannot be stored either.

The rest of the oil (that is, any not available in storage) must be bought in quantities to meet the blending requirements. The price of each kind of oil varies over the six-month period.

The two categories of oil cannot be refined on the same production line. There is a limit on how much oil of each category (vegetable or non vegetable) can be refined in a given month:

- Not more than 200 tons of vegetable oil can be refined per month.
- Not more than 250 tons of non vegetable oil can be refined per month.

There are constraints on the blending of oils:

- The product cannot blend more than three oils.
- When a given type of oil is blended into the product, at least 20 tons of that type must be used.
- If either vegetable oil 1 (v1) or vegetable oil 2 (v2) is blended in the product, then non vegetable oil 3 (o3) must also be blended in that product.

The final product (refined and blended) sells for a known price: 150 monetary units per ton.

The aim of the six-month plan is to minimize production and storage costs while maximizing profit.

Representing the Data

To represent the problem accurately, there are several questions to consider:

- *What Is Known?* on page 420
- *What Is Unknown?* on page 421
- *What Are the Constraints?* on page 421
- *What Is the Objective?* on page 423

What Is Known?

In this particular example, the planning period is six months, and there are five kinds of oil to be blended. Those details are represented as constants, like this:

```
const IloInt nbMonths   = 6;
const IloInt nbProducts = 5;
```

The five kinds of oil (vegetable and non vegetable) are represented by an enumeration, like this:

```
typedef enum { v1, v2, o1, o2, o3 } Product;
```

The varying price of the five kinds of oil over the six-month planning period is represented in a numeric matrix, like this:

```
NumMatrix cost(env, nbMonths);
cost[0]=IloNumArray(env, nbProducts, 110.0, 120.0, 130.0, 110.0, 115.0);
cost[1]=IloNumArray(env, nbProducts, 130.0, 130.0, 110.0,  90.0, 115.0);
cost[2]=IloNumArray(env, nbProducts, 110.0, 140.0, 130.0, 100.0,  95.0);
cost[3]=IloNumArray(env, nbProducts, 120.0, 110.0, 120.0, 120.0, 125.0);
cost[4]=IloNumArray(env, nbProducts, 100.0, 120.0, 150.0, 110.0, 105.0);
cost[5]=IloNumArray(env, nbProducts,  90.0, 100.0, 140.0,  80.0, 135.0);
```

That matrix could equally well be filled by data read from a file in a large-scale application.

What Is Unknown?

The variables of the problem can be represented in arrays:

- How much blended, refined oil to produce per month?
- How much raw oil to use per month?
- How much raw oil to buy per month?
- How much raw oil to store per month?

like this:

```
IloNumVarArray produce(env, nbMonths, 0, IloInfinity);
NumVarMatrix   use(env, nbMonths);
NumVarMatrix   buy(env, nbMonths);
NumVarMatrix   store(env, nbMonths);
IloInt i, p;
for (i = 0; i < nbMonths; i++) {
   use[i]   = IloNumVarArray(env, nbProducts, 0, IloInfinity);
   buy[i]   = IloNumVarArray(env, nbProducts, 0, IloInfinity);
   store[i] = IloNumVarArray(env, nbProducts, 0, 1000);
}
IloExpr profit(env);
```

Notice that how much to use and buy is initially unknown, and thus has an infinite upper bound, whereas the amount of oil that can be stored is limited, as you know from the description of the problem. Consequently, one of the constraints is expressed here as the upper bound of 1000 on the amount of oil by type that can be stored per month.

What Are the Constraints?

As you know from *Describing the Problem* on page 419, there are a variety of constraints in this problem.

For each type of oil, there must be 500 tons in storage at the end of the plan. That idea can be expressed like this:

```
for (p = 0; p < nbProducts; p++) {
  store[nbMonths-1][p].setBounds(500, 500);
}
```

The constraints on production in each month can all be expressed as statements in a for-loop:

- Not more than 200 tons of vegetable oil can be refined.

```
model.add(use[i][v1] + use[i][v2] <= 200);
```

- Not more than 250 tons of non-vegetable oil can be refined.

```
model.add(use[i][o1] + use[i][o2] + use[i][o3] <= 250);
```

- A blend cannot use more than three oils; or equivalently, of the five oils, two cannot be used in a given blend.

```
model.add((use[i][v1] == 0) +
          (use[i][v2] == 0) +
          (use[i][o1] == 0) +
          (use[i][o2] == 0) +
          (use[i][o3] == 0) >= 2);
```

- Blends composed of vegetable oil 1 (v1) or vegetable oil 2 (v2) must also include non vegetable oil 3 (o3).

```
model.add(IloIfThen(env, (use[i][v1] >= 20) || (use[i][v2] >= 20),
                         use[i][o3] >= 20));
```

- The constraint that if an oil is used at all in a blend, at least 20 tons of it must be used is expressed like this:

```
for (p = 0; p < nbProducts; p++)
            model.add((use[i][p] == 0) || (use[i][p] >= 20));
```

> ***Note:*** *If there were an upper bound on use, as well as this lower bound of 20, then this logical constraint could be replaced by semi-continous variables.*

- The fact that a limited amount of raw oil can be stored for later use is expressed like this:

```
if (i == 0) {
   for (IloInt p = 0; p < nbProducts; p++)
      model.add(500 + buy[i][p] == use[i][p] + store[i][p]);
}
else {
   for (IloInt p = 0; p < nbProducts; p++)
     model.add(store[i-1][p] + buy[i][p] ==
                  use[i][p] + store[i][p]);
}
```

What Is the Objective?

On a monthly basis, the profit can be represented as the sale price per ton (150) multiplied by the amount produced minus the cost of production and storage, like this:

```
profit += 150 * produce[i] - IloScalProd(cost[i],
                                         buy[i]) - 5 * IloSum(store[i]);
```

Developing the Model

First, create the model, like this:

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

Then use a for-loop to add the constraints for each month (from *What Are the Constraints?* on page 421), like this:

```
for (i = 0; i < nbMonths; i++) {
   model.add(use[i][v1] + use[i][v2] <= 200);
   model.add(use[i][o1] + use[i][o2] + use[i][o3] <= 250);
   model.add(3 * produce[i] <=
             8.8 * use[i][v1] + 6.1 * use[i][v2] +
             2   * use[i][o1] + 4.2 * use[i][o2] + 5 * use[i][o3]);
   model.add(8.8 * use[i][v1] + 6.1 * use[i][v2] +
             2   * use[i][o1] + 4.2 * use[i][o2] + 5 * use[i][o3]
             <= 6 * produce[i]);
   model.add(produce[i] == IloSum(use[i]));
   if (i == 0) {
      for (IloInt p = 0; p < nbProducts; p++)
         model.add(500 + buy[i][p] == use[i][p] + store[i][p]);
   }
   else {
      for (IloInt p = 0; p < nbProducts; p++)
        model.add(store[i-1][p] + buy[i][p] == use[i][p] + store[i][p]);
   }
   profit += 150 * produce[i] - IloScalProd(cost[i], buy[i]) -
             5 * IloSum(store[i]);

   model.add((use[i][v1] == 0) + (use[i][v2] == 0) + (use[i][o1] == 0) +
             (use[i][o2] == 0) + (use[i][o3] == 0) >= 2);
   for (p = 0; p < nbProducts; p++)
      model.add((use[i][p] == 0) || (use[i][p] >= 20));
   model.add(IloIfThen(env, (use[i][v1] >= 20) || (use[i][v2] >= 20),
     use[i][o3] >= 20));
}
```

To consolidate the monthly objectives, add the overall objective to the model, like this:

```
model.add(IloMaximize(env, profit));
```

Using Logical Constraints

You have already seen how to represent the logical constraints of this problem in *What Are the Constraints?* on page 421. However, they deserve a second glance because they illustrate an important point about logical constraints and automatic transformation in ILOG CPLEX.

```
// Logical constraints
         // The food cannot use more than 3 oils
         // (or at least two oils must not be used)
         model.add((use[i][v1] == 0) + (use[i][v2] == 0) + (use[i][o1] == 0) +
                   (use[i][o2] == 0) + (use[i][o3] == 0) >= 2);
         // When an oil is used, the quantity must be at least 20 tons
         for (p = 0; p < nbProducts; p++)
            model.add((use[i][p] == 0) || (use[i][p] >= 20));
         // If products v1 or v2 are used, then product o3 is also used
         model.add(IloIfThen(env, (use[i][v1] >= 20) || (use[i][v2] >= 20),
           use[i][o3] >= 20));
```

Consider, for example, the constraint that the blended product cannot use more than three oils in a batch. Given that constraint, many programmers might naturally write the following statement (or something similar) in C++:

```
model.add ( (use[i][v1] != 0)
          + (use[i][v2] != 0)
          + (use[i][o1] != 0)
          + (use[i][o2] != 0)
          + (use[i][o3] != 0)
          <= 3);
```

That statement expresses the same constraint without changing the set of solutions to the problem. However, the formulations are different and can lead to different running times and different amounts of memory used for the search tree. In other words, given a logical Engish expression, there may be more than one logical constraint for expressing it, and the different logical constraints may perform differently in terms of computing time and memory.

Logical Constraints in Optimization on page 411 introduced overloaded logical operators that you can use to combine linear, semi-continuous, or piecewise linear constraints in ILOG CPLEX. In this example, notice the overloaded logical operators ==, >=, +, || that appear in these logical constraints.

Solving the Problem

The following statement solves the problem to optimality:

```
if (cplex.solve()) {
```

These lines (the action of the if-statement) display the solution:

```
         cout << " Maximum profit = " << cplex.getObjValue() << endl;
         for (IloInt i = 0; i < nbMonths; i++) {
            IloInt p;
            cout << " Month " << i << " " << endl;
            cout << "  . buy   ";
            for (p = 0; p < nbProducts; p++) {
               cout << cplex.getValue(buy[i][p]) << "\t ";
            }
            cout << endl;
            cout << "  . use   ";
            for (p = 0; p < nbProducts; p++) {
               cout << cplex.getValue(use[i][p]) << "\t ";
            }
            cout << endl;
            cout << "  . store ";
            for (p = 0; p < nbProducts; p++) {
               cout << cplex.getValue(store[i][p]) << "\t ";
            }
            cout << endl;
         }
      }
      else {
         cout << " No solution found" << endl;
```

Ending the Program

Like other C++ applications using ILOG CPLEX with Concert Technology, this one ends with a call to free the memory used by the environment.

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

Example: foodmanufact.cpp

```
// --------------------------------------------------------------------------
// File: examples/src/foodmanufact.cpp
// Version 9.0
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------

#include <ilcplex/ilocplex.h>

ILOSTLBEGIN
```

```
typedef IloArray<IloNumVarArray> NumVarMatrix;
typedef IloArray<IloNumArray>    NumMatrix;

typedef enum { v1, v2, o1, o2, o3 } Product;
const IloInt nbMonths   = 6;
const IloInt nbProducts = 5;

int main() {
   IloEnv env;
   try {
      NumMatrix cost(env, nbMonths);
      cost[0]=IloNumArray(env, nbProducts, 110.0, 120.0, 130.0, 110.0, 115.0);
      cost[1]=IloNumArray(env, nbProducts, 130.0, 130.0, 110.0,  90.0, 115.0);
      cost[2]=IloNumArray(env, nbProducts, 110.0, 140.0, 130.0, 100.0,  95.0);
      cost[3]=IloNumArray(env, nbProducts, 120.0, 110.0, 120.0, 120.0, 125.0);
      cost[4]=IloNumArray(env, nbProducts, 100.0, 120.0, 150.0, 110.0, 105.0);
      cost[5]=IloNumArray(env, nbProducts,  90.0, 100.0, 140.0,  80.0, 135.0);

      IloNumVarArray produce(env, nbMonths, 0, IloInfinity);
      NumVarMatrix   use(env, nbMonths);
      NumVarMatrix   buy(env, nbMonths);
      NumVarMatrix   store(env, nbMonths);
      IloInt i, p;
      for (i = 0; i < nbMonths; i++) {
         use[i]   = IloNumVarArray(env, nbProducts, 0, IloInfinity);
         buy[i]   = IloNumVarArray(env, nbProducts, 0, IloInfinity);
         store[i] = IloNumVarArray(env, nbProducts, 0, 1000);
      }
      IloExpr profit(env);

      IloModel model(env);

      for (p = 0; p < nbProducts; p++) {
        store[nbMonths-1][p].setBounds(500, 500);
      }

      for (i = 0; i < nbMonths; i++) {
         model.add(use[i][v1] + use[i][v2] <= 200);
         model.add(use[i][o1] + use[i][o2] + use[i][o3] <= 250);
         model.add(3 * produce[i] <=
                   8.8 * use[i][v1] + 6.1 * use[i][v2] +
                   2   * use[i][o1] + 4.2 * use[i][o2] + 5 * use[i][o3]);
         model.add(8.8 * use[i][v1] + 6.1 * use[i][v2] +
                   2   * use[i][o1] + 4.2 * use[i][o2] + 5 * use[i][o3]
                   <= 6 * produce[i]);
         model.add(produce[i] == IloSum(use[i]));
         if (i == 0) {
            for (IloInt p = 0; p < nbProducts; p++)
               model.add(500 + buy[i][p] == use[i][p] + store[i][p]);
         }
         else {
            for (IloInt p = 0; p < nbProducts; p++)
              model.add(store[i-1][p] + buy[i][p] == use[i][p] + store[i][p]);
```

```
            }
            profit += 150 * produce[i] - IloScalProd(cost[i], buy[i]) -
                      5 * IloSum(store[i]);

            model.add((use[i][v1] == 0) + (use[i][v2] == 0) + (use[i][o1] == 0) +
                      (use[i][o2] == 0) + (use[i][o3] == 0) >= 2);
            for (p = 0; p < nbProducts; p++)
               model.add((use[i][p] == 0) || (use[i][p] >= 20));
            model.add(IloIfThen(env, (use[i][v1] >= 20) || (use[i][v2] >= 20),
              use[i][o3] >= 20));
         }

         model.add(IloMaximize(env, profit));

         IloCplex cplex(model);

         if (cplex.solve()) {
            cout << " Maximum profit = " << cplex.getObjValue() << endl;
            for (IloInt i = 0; i < nbMonths; i++) {
               IloInt p;
               cout << " Month " << i << " " << endl;
               cout << "  . buy   ";
               for (p = 0; p < nbProducts; p++) {
                  cout << cplex.getValue(buy[i][p]) << "\t ";
               }
               cout << endl;
               cout << "  . use   ";
               for (p = 0; p < nbProducts; p++) {
                  cout << cplex.getValue(use[i][p]) << "\t ";
               }
               cout << endl;
               cout << "  . store ";
               for (p = 0; p < nbProducts; p++) {
                  cout << cplex.getValue(store[i][p]) << "\t ";
               }
               cout << endl;
            }
         }
         else {
            cout << " No solution found" << endl;
         }
      }
      catch (IloException& ex) {
         cerr << "Error: " << ex << endl;
      }
      catch (...) {
         cerr << "Error" << endl;
      }
      env.end();
      return 0;
   }
```

19

Early Tardy Scheduling

This chapter shows you one way of using ILOG CPLEX to solve a scheduling problem. In it, you will see how to use logical constraints, piecewise linear functions, and aggressive MIP emphasis.

- *Describing the Problem* on page 430
- *Understanding the Data File* on page 430
- *Reading the Data* on page 431
- *Creating Variables* on page 431
- *Stating Precedence Constraints* on page 432
- *Stating Resource Constraints* on page 432
- *Representing the Piecewise Linear Cost Function* on page 433
- *Transforming the Problem* on page 433
- *Solving the Problem* on page 434
- *Example: etsp.cpp* on page 434

Note: *These logical constraint facilities are available in ILOG CPLEX only with Concert Technology for C++ users.*

Describing the Problem

The problem is to schedule a number of jobs over a group of resources. In this context, a job is a set of activities that must be carried out, one after another. Each resource can process only one single activity at a time.

For each job, there is a due date, that is, the ideal date to finish this job by finishing the last activity of this job. If the job is finished earlier than the due date, there will be a cost proportional to the earliness. Symmetrically, if the job is finished later than the due date, there will be a cost proportional to the tardiness.

As "just in time" inventory management becomes more and more important, problems like this occur more frequently in industrial settings.

Understanding the Data File

The data for this problem are available online with your installation of the product in the file *yourCPLEXhome*`/examples/data/etsp.dat`.

In this particular example, the number of activities of a job matches the number of resources. However, you can easily modify the application to accommodate a different number of jobs and resources.

In the file for this particular example, the first number (10) indicates the number of jobs and resources and thus the number of activities per job. Each of the other lines of the file corresponds to the description of a job. In a line, each pair of numbers corresponds to one activity. In each such pair, the first number represents the index of the resource that must process this activity. The second number in each pair indicates the duration of the activity. For example, if you look at `job0`, you see:

```
2 41 0 32 4 72 1 65 3 53 9 99 7 35 8 46 5 53 6 2 983 3 10
```

That line means that `activity0` of `job0` must be processed by `resource2` and it lasts 41 units of time, `activity1` of this job must be processed on `resource0` and lasts 32 time units, and so forth.

The last three numbers of a line indicate the due date, the earliness cost, and the tardiness cost. For example, if `job0` finishes by date 983, then no costs are incurred. If it finishes at 981, then it incurs an earliness cost of (3x2) = 6. If it finished at 985, then it incurs a tardiness cost of 20.

Reading the Data

As you see in Code Sample 19.1, the first part of this application reads data from a file and fills two 10x10 matrices:

- `resourceIndex` contains the indices of the resources;
- `duration` contains the duration of each activity.

```
IloNumArray jobDueDate(env, nbJob);
IloNumArray jobEarlinessCost(env, nbJob);
IloNumArray jobTardinessCost(env, nbJob);

NumMatrix   duration(env, nbJob);
for(j = 0; j < nbJob; j++) {
  duration[j] = IloNumArray(env, nbResource);
for (i = 0; i < nbResource; i++) {
    IloInt resourceIndex;
    f >> resourceIndex;
    f >> duration[j][i];
    activityOnAResource[resourceIndex].add(i);
  }
  f >> jobDueDate[j];
  f >> jobEarlinessCost[j];
  f >> jobTardinessCost[j];
}
```

Code Sample 19.1 *Reading data for early tardy scheduling*

Each line in the data file corresponds to an array in the matrix and thus represents all the information about activities for a given job.

For each job, other arrays contain further information from the data file:

- `jobDueDate` contains the due date for each job;
- `jobEarlinessCost` contains the penalty for being too early for each job;
- `jobTardinessCost` contains the penalty for being too late for each job.

The matrix `activityOnAResource` contains the sets of activities that must be scheduled on the same resource. This information will be used to state resource constraints.

```
IntMatrix   activityOnAResource(env, nbResource);
    for(i = 0; i < nbResource; i++) {
      activityOnAResource[i] = IloIntArray(env);
    }
```

Creating Variables

The unknowns of the problem are the starting dates of the various activities. To represent these dates with Concert Technology modeling objects, the application creates a matrix of

numeric variables (that is, instances of `IloNumVar`) with bounds between 0 and `Horizon`, where `Horizon` is the maximum starting date for an activity that does not exclude interesting solutions of the problem. In this example, it is set arbitrarily at 10000.

```
NumVarMatrix s(env, nbJob);
for(j = 0; j < nbJob; j++){
  s[j] = IloNumVarArray(env, nbResource, 0, Horizon);
}
```

Stating Precedence Constraints

In each job, activities must be processed one after the other. This order is enforced by the precedence constraints, which look like this:

```
for(j = 0; j < nbJob; j++){
     for(i = 1; i < nbResource; i++){
       model.add(s[j][i] >= s[j][i-1] + duration[j][i-1]);
     }
   }
```

Stating Resource Constraints

Each resource can process one activity at a time. To avoid having two (or more) activities that share the same resource overlap with each other, disjunctive constraints are added to the model. Disjunctive constraints look like this:

```
s1 >= s2 + d2 or s2 >= s1 + d1
```

where `s` is the starting date of an activity and `d` is its duration.

If `n` activities need to be processed on the same resource then about `(n*n)/2` disjunctions need to be stated and added to the model, like this:

```
for(i = 0; i < nbResource; i++) {
     IloInt end = nbJob - 1;
     for(j = 0; j < end; j++){
       IloInt a = activityOnAResource[i][j];
       for(IloInt k = j + 1; k < nbJob; k++){
         IloInt b = activityOnAResource[i][k];
         model.add(s[j][a] >= s[k][b] + duration[k][b]
                   ||
                   s[k][b] >= s[j][a] + duration[j][a]);
       }
     }
   }
```

Representing the Piecewise Linear Cost Function

The earliness-tardiness cost function is the sum of piecewise linear functions having two segments, as you see in Figure 19.1. The function takes as an argument the completion date of the last activity of a job (in other words, the starting date plus the duration). In that two-segment function, the slope of the first segment is (-1) times the earliness cost, and the slope of the second segment is the tardiness cost. Moreover, at the due date, the cost is zero. Consequently, the function can be represented as a piecewise linear function with one breakpoint and two slopes, like this:

```
IloInt last = nbResource - 1;
   IloExpr costSum(env);
   for(j = 0; j < nbJob; j++) {
     costSum += IloPiecewiseLinear(s[j][last] + duration[j][last],
                                   IloNumArray(env, 1, jobDueDate[j]),
                                   IloNumArray(env,
                                               2,
                                               -jobEarlinessCost[j],
                                               jobTardinessCost[j]),
                                   jobDueDate[j], 0);
   }
   model.add(IloMinimize(env, costSum));
```

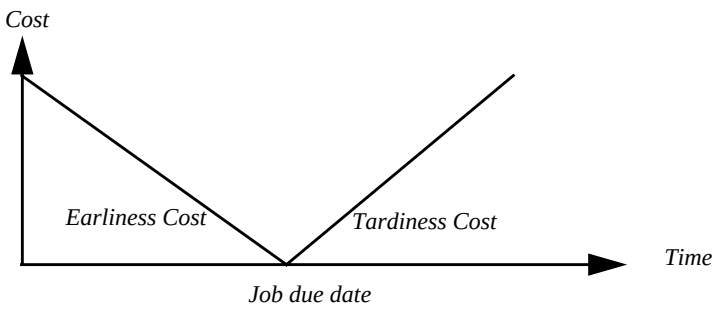

Figure 19.1 *Earliness and Tardiness as Piecewise Linear Cost Function*

Transforming the Problem

When ILOG CPLEX extracts disjunctive constraints and piecewise linear functions, it transforms them to produce a MIP with linear constraints and possibly SOS constraints over integer or continuous variables. The tightness of the transformation depends on the bounds set on the variables.

In this example, the `Horizon` is set to 10000, but if you have information about your problem that indicates that a good or even optimal solution exists with a tighter horizon (say,

2000 instead) then the linear formulation of disjunctions will be tighter with that tighter horizon.

That kind of tightening often leads to a better lower bound at the root node and to a reduction of the solving time.

It is also important to note that the piecewise linear expressions in this example are convex and their sum needs to be minimized. Therefore, this part of the problem does not lead to a MIP formulation, but instead it gives rise to an LP. This advantage is detected automatically by ILOG CPLEX, and it does not introduce any integer variable to transform this part of the problem.

Solving the Problem

An emphasis on finding hidden feasible solutions has proven particularly for this problem so this example makes that selection by setting the `MIPEmphasis` parameter to 4.

Example: etsp.cpp

You can see the entire example here or online in the standard distribution of ILOG CPLEX at *yourCPLEXhome*`/examples/src/etsp.cpp`.

```
// -------------------------------------------------- -*- C++ -*-
// File: examples/src/etsp.cpp
// Version 9.0
// ---------------------------------------------------------------
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ---------------------------------------------------------------
//

#include <ilcplex/ilocplex.h>

ILOSTLBEGIN

typedef IloArray<IloIntArray>    IntMatrix;
typedef IloArray<IloNumArray>    NumMatrix;
typedef IloArray<IloNumVarArray> NumVarMatrix;

const IloInt Horizon = 10000;

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

```
IloInt i, j;

const char* filename;
if (argc > 1)
  filename = argv[1];
else
  filename = "../../../examples/data/etsp.dat";
ifstream f(filename, ios::in);
if (!f) {
  cerr << "No such file: " << filename << endl;
  throw(1);
}

IloEnv env;

IloInt nbJob;
IloInt nbResource;

f >> nbJob;
nbResource = nbJob;

// Activity processed by a resource
IntMatrix   activityOnAResource(env, nbResource);
for(i = 0; i < nbResource; i++) {
  activityOnAResource[i] = IloIntArray(env);
}

IloNumArray jobDueDate(env, nbJob);        // Due date of a job
IloNumArray jobEarlinessCost(env, nbJob);  // Earliness cost of a job
IloNumArray jobTardinessCost(env, nbJob);  // Tardiness cost of a job

NumMatrix   duration(env, nbJob);          // Duration for an activity
for(j = 0; j < nbJob; j++) {
  duration[j] = IloNumArray(env, nbResource);
  // Read jobs duration
  for (i = 0; i < nbResource; i++) {
    IloInt resourceIndex;
    f >> resourceIndex;
    f >> duration[j][i];
    activityOnAResource[resourceIndex].add(i);
  }
  // Read cost information
  f >> jobDueDate[j];
  f >> jobEarlinessCost[j];
  f >> jobTardinessCost[j];
}

IloModel model(env);

// Create start variables
NumVarMatrix s(env, nbJob);
for(j = 0; j < nbJob; j++){
  s[j] = IloNumVarArray(env, nbResource, 0, Horizon);
```

```
  }

  // State precedence constraints
  for(j = 0; j < nbJob; j++){
    for(i = 1; i < nbResource; i++){
      model.add(s[j][i] >= s[j][i-1] + duration[j][i-1]);
    }
  }

  // State disjunctive constraints for each resource
  for(i = 0; i < nbResource; i++) {
    IloInt end = nbJob - 1;
    for(j = 0; j < end; j++){
      IloInt a = activityOnAResource[i][j];
      for(IloInt k = j + 1; k < nbJob; k++){
        IloInt b = activityOnAResource[i][k];
        model.add(s[j][a] >= s[k][b] + duration[k][b]
                  ||
                  s[k][b] >= s[j][a] + duration[j][a]);
      }
    }
  }

  // The cost is the sum of earliness or tardiness cost of each job
  IloInt last = nbResource - 1;
  IloExpr costSum(env);
  for(j = 0; j < nbJob; j++) {
    costSum += IloPiecewiseLinear(s[j][last] + duration[j][last],
                                  IloNumArray(env, 1, jobDueDate[j]),
                                  IloNumArray(env, 2,
                                              -jobEarlinessCost[j],
                                               jobTardinessCost[j]),
                                  jobDueDate[j], 0);
  }
  model.add(IloMinimize(env, costSum));

  IloCplex cplex(env);

  cplex.extract(model);
  cplex.setParam(IloCplex::MIPEmphasis, 4);

  if (cplex.solve()){
    cout << " Optimal Value = " << cplex.getObjValue() << endl;
  }
}
catch (IloException& ex) {
  cerr << "Error: " << ex << endl;
 }
 catch (...) {
   cerr << "Error" << endl;
}
env.end();
return 0;
```

```
}
```

Part V

Advanced Programming Techniques

This part documents advanced programming techniques for users of ILOG CPLEX. It shows you how to apply query routines to gather information while ILOG CPLEX is working. It demonstrates how to redirect the search with goals or callbacks. This part also covers user-defined constraints and pools of lazy constraints. It documents the advanced MIP control interface and the advanced aspects of preprocessing: presolve and aggregation. It also introduces special considerations about parallel programming with ILOG CPLEX.

This part of the manual assumes that you are already familiar with earlier parts of the manual. It contains:

- *Using Query Routines in the Callable Library* on page 441
- *Using Goals* on page 453
- *Using Callbacks* on page 477
- *Goals and Callbacks: a Comparison* on page 505
- *User Cut and Lazy Constraint Pools* on page 507
- *Advanced MIP Control Interface* on page 513
- *Advanced Presolve Routines* on page 521
- *Parallel Optimizers* on page 531

C H A P T E R

20

Using Query Routines in the Callable Library

This chapter tells you how to use the Callable Library query routines. In Concert Technology (for C++ and Java users), query methods such as `getName` use objects that can be referenced directly. Thus this chapter concentrates on the Callable Library routines only. However, equivalent Concert Technology examples are also provided in the distribution. This chapter includes these sections:

- *Using Surplus Arguments for Array Allocations* on page 441;
- *Example: Using Query Routines lpex7.c* on page 443.

Using Surplus Arguments for Array Allocations

Most of the ILOG CPLEX query routines in the Callable Library require your application to allocate memory for one or more arrays that will contain the results of the query. In many cases, your application—the calling program—does not know the size of these arrays in advance. For example, in a call to `CPXgetcols` requesting the matrix data for a range of columns, your application needs to pass the arrays `cmatind` and `cmatval` for ILOG CPLEX to populate with matrix coefficients and row indices. However, unless your application has carefully kept track of the number of nonzeros in each column throughout

the problem specification and, if applicable, throughout its modification, the actual length of these arrays remains unknown.

Fortunately, the ILOG CPLEX query routines in the Callable Library contain a `surplus_p` argument that, when used in conjunction with the array length arguments, enables you first to call the query routine to determine the length of the required array. Then, when the length is known, your application can properly allocate these arrays. Afterwards, your application makes a second call to the query routine with the correct array lengths to obtain the requested data.

For example, consider a program that needs to call `CPXgetcols` to access a range of columns. Here is the list of arguments for `CPXgetcols`.

```
CPXgetcols (CPXENVptr env,
            CPXLPptr lp,
            int *nzcnt_p,
            int *cmatbeg,
            int *cmatind,
            double *cmatval,
            int cmatspace,
            int *surplus_p,
            int begin,
            int end);
```

The arrays `cmatind` and `cmatval` require one element for each nonzero matrix coefficient in the requested range of columns. The required length of these arrays, specified in `cmatspace`, remains unknown at the time of the query. Your application—the calling program—can determine the length of these arrays by first calling `CPXgetcols` with a value of `0` for `cmatspace`. This call will return an error status of `CPXERR_NEGATIVE_SURPLUS` indicating a shortfall of the array length specified in `cmatspace` (in this case, `0`); it will also return the actual number of matrix nonzeros in the requested range of columns. `CPXgetcols` deposits this shortfall as a negative number in the integer pointed to by `surplus_p`. Your application can then negate this shortfall and allocate the arrays `cmatind` and `cmatval` sufficiently long to contain all the requested matrix elements.

The following sample of code illustrates this procedure. The first call to `CPXgetcols` passes a value of `0` for `cmatspace` in order to obtain the shortfall in `cmatsz`. The sample then uses

the shortfall to allocate the arrays `cmatind` and `cmatval` properly; then it calls `CPXgetcols` again to obtain the actual matrix coefficients and row indices.

```
status = CPXgetcols (env, lp, &nzcnt, cmatbeg, NULL, NULL,
                     0, &cmatsz, 0, numcols - 1);
if ( status != CPXERR_NEGATIVE_SURPLUS ) {
   if ( status != 0 ) {
      CPXmsg (cpxerror,
              "CPXgetcols for surplus failed, status = %d\n", status);
      goto TERMINATE;
   }
   CPXmsg (cpxwarning,
           "All columns in range [%d, %d] are empty.\n",
           0, (numcols - 1));
}
cmatsz   = -cmatsz;
cmatind  = (int *) malloc ((unsigned) (1 + cmatsz)*sizeof(int));
cmatval  = (double *) malloc ((unsigned) (1 + cmatsz)*sizeof(double));
if ( cmatind == NULL || cmatval == NULL ) {
   CPXmsg (cpxerror, "CPXgetcol mallocs failed\n");
   status = 1;
   goto TERMINATE;
}
status = CPXgetcols (env, lp, &nzcnt, cmatbeg, cmatind, cmatval,
                     cmatsz, &surplus, 0, numcols - 1);
if ( status ) {
   CPXmsg (cpxerror, "CPXgetcols failed, status = %d\n", status);
   goto TERMINATE;
}
```

That sample code (or your application) does not need to determine the length of the array `cmatbeg`. The array `cmatbeg` has one element for each column in the requested range. Since this length is known ahead of time, your application does not need to call a query routine to calculate it. More generally, query routines use surplus arguments in this way only for the length of any array required to store problem data of unknown length. Problem data in this category include nonzero matrix entries, row and column names, other problem data names, special ordered sets (SOS), priority orders, and MIP start information.

Example: Using Query Routines lpex7.c

This example uses the ILOG CPLEX Callable Library *query routine* `CPXgetcolname` to get the column names from a problem object. To do so, it applies the programming pattern just outlined in *Using Surplus Arguments for Array Allocations* on page 441. It derives from the example `lpex2.c` from the ILOG CPLEX *Getting Started* manual. This query-routine example differs from that simpler example in several ways:

- The example calls `CPXgetcolname` twice after optimization: the first call determines how much space to allocate to hold the names; the second call gets the names and stores them in the arrays `cur_colname` and `cur_colnamestore`.

- When the example prints its answer, it uses the names as stored in `cur_colname`. If no names exist there, the example creates generic names.

This example assumes that the current problem has been read from a file by `CPXreadcopyprob`. You can adapt the example to use other ILOG CPLEX query routines to get information about any problem read from a file.

The complete program, `lpex7.c`, appears here and online in the standard distribution.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/lpex7.c                                              */
/*  Version 9.0                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2003 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* lpex7.c - Reading in and optimizing a problem.  Printing
             names with the answer.  This is a modification of
             lpex2.c  */

/* To run this example, command line arguments are required.
   i.e.,   lpex7   filename   method
   where
       filename is the name of the file, with .mps, .lp, or .sav extension
       method   is the optimization method
                   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
   Example:
       lpex7  example.mps  o
 */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

/* Bring in the declarations for the string and character functions
   and malloc */

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

/* Include declarations for functions in this program */
```

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


int
main (int argc, char *argv[])
{
   /* Declare and allocate space for the variables and arrays where we will
      store the optimization results including the status, objective value,
      maximum bound violation, variable values, and basis. */

   int      solnstat, solnmethod, solntype;
   double   objval, maxviol;
   double   *x     = NULL;
   int      *cstat = NULL;
   int      *rstat = NULL;

   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status = 0;
   int           j;
   int           cur_numrows, cur_numcols;
   char          **cur_colname = NULL;
   char          *cur_colnamestore = NULL;
   int           cur_colnamespace;
   int           surplus;
   int           method;

   char          *basismsg;

   /* Check the command line arguments */

   if (( argc != 3 )                                 ||
       ( strchr ("podhbnsc", argv[2][0]) == NULL )  ) {
      usage (argv[0]);
      goto TERMINATE;
   }

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

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

   if ( env == NULL ) {
```

```
   char  errmsg[1024];
   fprintf (stderr, "Could not open CPLEX environment.\n");
   CPXgeterrorstring (env, status, errmsg);
   fprintf (stderr, "%s", errmsg);
   goto TERMINATE;
}

/* Turn on output to the screen */

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

/* Create the problem, using the filename as the problem name */

lp = CPXcreateprob (env, &status, argv[1]);

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

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

/* Now read the file, and copy the data into the created lp */

status = CPXreadcopyprob (env, lp, argv[1], NULL);
if ( status ) {
   fprintf (stderr, "Failed to read and copy the problem data.\n");
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */

switch (argv[2][0]) {
   case 'o':
      method = CPX_ALG_AUTOMATIC;
      break;
   case 'p':
      method = CPX_ALG_PRIMAL;
      break;
   case 'd':
      method = CPX_ALG_DUAL;
      break;
```

```
      case 'n':
         method = CPX_ALG_NET;
         break;
      case 'h':
         method = CPX_ALG_BARRIER;
         break;
      case 'b':
         method = CPX_ALG_BARRIER;
         status = CPXsetintparam (env, CPX_PARAM_BARCROSSALG, CPX_ALG_NONE);
         if ( status ) {
            fprintf (stderr,
                     "Failed to set the crossover method, error %d.\n",
status);
            goto TERMINATE;
         }
         break;
      case 's':
         method = CPX_ALG_SIFTING;
         break;
      case 'c':
         method = CPX_ALG_CONCURRENT;
         break;
      default:
         method = CPX_ALG_NONE;
         break;
   }

   status = CPXsetintparam (env, CPX_PARAM_LPMETHOD, method);
   if ( status ) {
      fprintf (stderr,
               "Failed to set the optimization method, error %d.\n", status);
      goto TERMINATE;
   }

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

   solnstat = CPXgetstat (env, lp);

   if      ( solnstat == CPX_STAT_UNBOUNDED ) {
      printf ("Model is unbounded\n");
      goto TERMINATE;
   }
   else if ( solnstat == CPX_STAT_INFEASIBLE ) {
      printf ("Model is infeasible\n");
      goto TERMINATE;
   }
   else if ( solnstat == CPX_STAT_INForUNBD ) {
      printf ("Model is infeasible or unbounded\n");
      goto TERMINATE;
```

```
}

status = CPXsolninfo (env, lp, &solnmethod, &solntype, NULL, NULL);
if ( status ) {
   fprintf (stderr, "Failed to obtain solution info.\n");
   goto TERMINATE;
}
printf ("Solution status %d, solution method %d\n", solnstat, solnmethod);

if ( solntype == CPX_NO_SOLN ) {
   fprintf (stderr, "Solution not available.\n");
   goto TERMINATE;
}

status = CPXgetobjval (env, lp, &objval);
if ( status ) {
   fprintf (stderr, "Failed to obtain objective value.\n");
   goto TERMINATE;
}
printf ("Objective value %.10g.\n", objval);

/* The size of the problem should be obtained by asking CPLEX what
   the actual size is.  cur_numrows and cur_numcols store the
   current number of rows and columns, respectively.  */

cur_numcols = CPXgetnumcols (env, lp);
cur_numrows = CPXgetnumrows (env, lp);

/* Retrieve basis, if one is available */

if ( solntype == CPX_BASIC_SOLN ) {
   cstat = (int *) malloc (cur_numcols*sizeof(int));
   rstat = (int *) malloc (cur_numrows*sizeof(int));
   if ( cstat == NULL || rstat == NULL ) {
      fprintf (stderr, "No memory for basis statuses.\n");
      goto TERMINATE;
   }

   status = CPXgetbase (env, lp, cstat, rstat);
   if ( status ) {
      fprintf (stderr, "Failed to get basis; error %d.\n", status);
      goto TERMINATE;
   }
}
else {
   printf ("No basis available\n");
}


/* Retrieve solution vector */

x = (double *) malloc (cur_numcols*sizeof(double));
if ( x == NULL ) {
```

```
   fprintf (stderr, "No memory for solution.\n");
   goto TERMINATE;
}

status = CPXgetx (env, lp, x, 0, cur_numcols-1);
if ( status ) {
   fprintf (stderr, "Failed to obtain primal solution.\n");
   goto TERMINATE;
}


/* Now get the column names for the problem.  First we determine how
   much space is used to hold the names, and then do the allocation.
   Then we call CPXgetcolname() to get the actual names. */

status = CPXgetcolname (env, lp, NULL, NULL, 0, &surplus, 0,
                        cur_numcols-1);

if (( status != CPXERR_NEGATIVE_SURPLUS ) &&
    ( status != 0 )                         )  {
   fprintf (stderr,
            "Could not determine amount of space for column names.\n");
   goto TERMINATE;
}

cur_colnamespace = - surplus;
if ( cur_colnamespace > 0 ) {
   cur_colname      = (char **) malloc (sizeof(char *)*cur_numcols);
   cur_colnamestore = (char *)  malloc (cur_colnamespace);
   if ( cur_colname      == NULL ||
        cur_colnamestore == NULL   ) {
      fprintf (stderr, "Failed to get memory for column names.\n");
      status = -1;
      goto TERMINATE;
   }
   status = CPXgetcolname (env, lp, cur_colname, cur_colnamestore,
                           cur_colnamespace, &surplus, 0, cur_numcols-1);
   if ( status ) {
      fprintf (stderr, "CPXgetcolname failed.\n");
      goto TERMINATE;
   }
}
else {
   printf ("No names associated with problem.  Using Fake names.\n");
}

/* Write out the solution */

for (j = 0; j < cur_numcols; j++) {
   if ( cur_colnamespace > 0 ) {
      printf ("%-16s:  ", cur_colname[j]);
   }
```

```
      else {
         printf ("Fake%-6.6d      :  ", j);;
      }
      printf ("%17.10g", x[j]);
      if ( cstat != NULL ) {
         switch (cstat[j]) {
            case CPX_AT_LOWER:
               basismsg = "Nonbasic at lower bound";
               break;
            case CPX_BASIC:
               basismsg = "Basic";
               break;
            case CPX_AT_UPPER:
               basismsg = "Nonbasic at upper bound";
               break;
            case CPX_FREE_SUPER:
               basismsg = "Superbasic, or free variable at zero";
               break;
            default:
               basismsg = "Bad basis status";
               break;
         }
         printf ("  %s",basismsg);
      }
      printf ("\n");
   }

   /* Display the maximum bound violation. */

   status = CPXgetdblquality (env, lp, &maxviol, CPX_MAX_PRIMAL_INFEAS);
   if ( status ) {
      fprintf (stderr, "Failed to obtain bound violation.\n");
      goto TERMINATE;
   }
   printf ("Maximum bound violation = %17.10g\n", maxviol);

TERMINATE:

   /* Free up the basis and solution */

   free_and_null ((char **) &cstat);
   free_and_null ((char **) &rstat);
   free_and_null ((char **) &x);
   free_and_null ((char **) &cur_colname);
   free_and_null ((char **) &cur_colnamestore);

   /* Free up the problem, if necessary */

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

```
   }

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

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

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

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

   return (status);

}  /* END main */



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

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */


static void
usage (char *progname)
{
   fprintf (stderr,"Usage: %s filename algorithm\n", progname);
   fprintf (stderr,"   where filename is a file with extension \n");
   fprintf (stderr,"      MPS, SAV, or LP (lower case is allowed)\n");
   fprintf (stderr,"   and algorithm is one of the letters\n");
   fprintf (stderr,"      o          default\n");
   fprintf (stderr,"      p          primal simplex\n");
   fprintf (stderr,"      d          dual simplex\n");
   fprintf (stderr,"      n          network simplex\n");
   fprintf (stderr,"      b          barrier\n");
   fprintf (stderr,"      h          barrier with crossover\n");
```

```
   fprintf (stderr,"      s          sifting\n");
   fprintf (stderr,"      c          concurrent\n");
   fprintf (stderr," Exiting...\n");
} /* END usage */
```

C H A P T E R

21

Using Goals

This chapter explores goals and their role in a branch & cut search. In it, you will learn about:

- *Branch & Cut in ILOG CPLEX* on page 454
- *The Goal Stack* on page 460
- *Memory Management and Goals* on page 461
- *Cuts and Goals* on page 463
- *Injecting Heuristic Solutions* on page 468
- *Controlling Goal-Defined Search* on page 469
- *Search Limits* on page 475

Note: *Goals are implemented by IloCplex::Goal, not IloGoal as in other ILOG products.*

Branch & Cut in ILOG CPLEX

Goals allow you to take control of the branch & cut search procedure used by `IloCplex` to solve MIP problems. To help you understand how to use goals with `IloCplex`, this section reviews how this procedure works.

The `IloCplex` branch & cut search procedure manages a search tree consisting of nodes. Every node represents a subproblem to be solved, and the root node of the tree represents the entire problem. Nodes are called *active* if they have not yet been processed.

The tree is first initialized to contain the root node as the only active node. `IloCplex` processes active nodes from the tree until either no more active nodes are available or some limit has been reached. Once a node has been processed, it is no longer active.

When processing a node, `IloCplex` starts by solving the continuous relaxation of its subproblem—that is, the subproblem without integrality constraints. If the solution violates any cuts, `IloCplex` adds them to the node problem and re-solves. This is iterated until no more violated cuts are found by `IloCplex`. If at any point the relaxation becomes infeasible, the node is pruned, that is, it is removed from the tree.

To solve the node problem, `IloCplex` checks whether the solution satisfies the integrality constraints. If so, and if its objective value is better than that of the current incumbent, the solution of the node problem is used as the new incumbent. Otherwise, `IloCplex` splits the node problem into one or two smaller subproblems, typically by branching on a variable that violates its integrality condition. These subproblems are added to the tree as active nodes and the current node is deactivated.

The primary use of goals is to take control of these last steps, namely the integer-feasibility test and the creation of subproblems. However, as discussed later, goals also allow you to add local and global cuts.

> ***Note:*** *The discussion of the details of using goals will be presented mainly in terms of the C++ API. The Java and C#.NET APIs follow the same design and are thus equivalent at this level of discussion. In cases where a difference between these APIs needs to be observed, the point will be raised. Where the difference is only in syntax, the other syntax will be mentioned in parentheses following the C++ syntax.*

In C++, goals are implemented in objects of type `IloCplex::GoalI` (having handle class `IloCplex::Goal`). In Java, goals are implemented in objects of type `IloCplex.Goal` (and there are no handle classes). The method `IloCplex::GoalI::execute` (`IloCplex.Goal.execute`) is where all the control is implemented. This method is called by `IloCplex` after a node relaxation has been solved and all cuts have been added. Invoking method `execute` of a goal is often referred to as *executing a goal*. When the method `execute` is executed, other methods of the class `IloCplex::GoalI` (`IloCplex.Goal`)

can be called to query information about the current node problem and the solution of its relaxation.

Typically, the implementation of method `execute` will perform the following steps:

1. Check feasibility. An interesting possibility here is that the feasibility check may include more than verifying integrality of the solution. This allows you to enforce constraints that could not reasonably be expressed using linear constraints through cuts or branching. In other words, this allows you to use goals in a way that makes them part of the model to be solved. Such a use is common in Constraint Programming, but it is less frequently used in Mathematical Programming.
2. Optionally find local or global cuts to be added. Local cuts will be respected only for the subtree below the current node, whereas global cuts will be enforced for all nodes from then on.
3. Optionally construct a solution and pass it to `IloCplex`.
4. Instruct `IloCplex` how to proceed. Instructing `IloCplex` how to proceed is done through the return value of method `execute`, which is another goal. `IloCplex` simply continues by executing this goal.

`IloCplex` provides a selection of special goals that can be used to specify how to proceed:

- The method `IloCplex::GoalI::OrGoal` (`IloCplex.or`) returns a goal that creates subnodes of the current node. This function takes at least 2 and up to 6 goals as parameters. For each of its parameters, the `Or` goal will create a subnode in such a way that when processing that subnode, the corresponding goal will be executed. Once the goal has been executed, the current node is immediately deactivated.
- The method `IloCplex::GoalI::AndGoal` (`IloCplex.and`) also takes goals as parameters. It returns a goal that will cause `IloCplex` to execute the goals passed as parameters in the order of the parameters.
- The method `IloCplex::GoalI::FailGoal` (`IloCplex.failGoal`) creates a goal that causes `IloCplex` to prune the current node. In other words, it discontinues the search at the node where the goal is executed. `IloCplex` will continue with another active node from the tree, if available.
- Class `IloCplex::Goal` has constructors that take an `IloRange` object or an `IloRangeArray` (`IloRange[]`)object as parameters. When one of these constructors is used, a *local cut goal* is created. Local cut goals add local cuts to the node where they are executed. To create local cut goals with the Java API, use method `IloCplex.constraintGoal` or if more convenient, one of the methods `IloCplex.leGoal`, `IloCplex.geGoal` or `IloCplex.eqGoal`.

- The 0 (`null`) goal, or empty goal, that is, an `IloCplex::Goal` handle object with 0 implementation pointer can also be returned by method `IloCplex::GoalI::execute`. In most cases this will instruct `IloCplex` to take over control of the branch & cut search with its built-in strategies.

Since `IloCplex::GoalI::OrGoal` (`IloCplex.or`) and `IloCplex::GoalI::AndGoal` (`IloCplex.and`) take other goals as parameters, goals can be combined into aggregate goals. In fact, this is how goals are typically used for specifying a branching strategy. A typical return goal of a user-written `execute` method for C++ looks like this:

```
return AndGoal(OrGoal(var <= IloFloor(val), var >= IloFloor(val)+1), this);
```

and for Java looks like this:

```
return cplex.and(cplex.or(cplex.leGoal(var, Math.floor(val)),
                          cplex.geGoal(var, Math.floor(val)+1)), this);
```

For the C++ case, note that since this statement would be called from the `execute` method of a subclass of `IloCplex::GoalI`, the full method name `IloCplex::GoalI::OrGoal` can be abbreviated to `OrGoal` (and likewise `AndGoal`).

This return statement returns an `And` goal that first executes the `Or` goal and then the current goal itself specified by the `this` parameter. When the `Or` goal is executed next, it will create two subnodes. In the first subnode, the first local cut goal representing $\texttt{var} \le \lfloor \texttt{val} \rfloor$ (where $\lfloor \texttt{val} \rfloor$ denotes the floor of `val`) will be executed, thus adding the constraint $\texttt{var} \le \lfloor \texttt{val} \rfloor$ for the subtree of this node. Similarly, the second subnode will be created, and when executing its constraint goal the constraint $\texttt{var} \ge \lfloor \texttt{val} \rfloor + 1$ will be added for the subtree. `this` is then executed on each of the nodes that have just been created; the same goal is used for both subtrees. Further details on how goals are processed will be discussed later.

Here is an example that precedes further discussions of goals. This example is available as file `ilogoalex1.cpp` in the `examples/src` subdirectory of your ILOG CPLEX distribution. The equivalent Java version can be found as `GoalEx1.java` in the same location.

```
// ------------------------------------------------------------ -*- C++ -*-
// File: examples/src/ilogoalex1.cpp
// Version 9.0
// ---------------------------------------------------------------------------
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ---------------------------------------------------------------------------
//
// ilogoalex1.cpp - A simple goal example.
//
//                  Create a branch goal that chooses the variable
//                  with the largest objective from amongst those
```

```
//                  with the largest integer infeasibility.
//

#include <ilcplex/ilocplex.h>

ILOSTLBEGIN

static void usage (const char *progname);

// Branch on var with largest objective coefficient
// among those with largest infeasibility

ILOCPLEXGOAL1(MyBranchGoal, IloNumVarArray, vars) {
   IloNumArray x;
   IloNumArray obj;
   IntegerFeasibilityArray feas;

   x    = IloNumArray(getEnv());
   obj  = IloNumArray(getEnv());
   feas = IntegerFeasibilityArray(getEnv());
   getValues(x, vars);
   getObjCoefs(obj, vars);
   getFeasibilities(feas, vars);

   IloInt bestj  = -1;
   IloNum maxinf = 0.0;
   IloNum maxobj = 0.0;
   IloInt cols = vars.getSize();
   for (IloInt j = 0; j < cols; ++j) {
      if ( feas[j] == Infeasible ) {
         IloNum xj_inf = x[j] - IloFloor (x[j]);
         if ( xj_inf > 0.5 )
            xj_inf = 1.0 - xj_inf;
         if ( xj_inf >= maxinf                                &&
             (xj_inf > maxinf || IloAbs (obj[j]) >= maxobj)  ) {
            bestj  = j;
            maxinf = xj_inf;
            maxobj = IloAbs (obj[j]);
         }
      }
   }

   IloCplex::Goal res;
   if ( bestj >= 0 ) {
      res = AndGoal(OrGoal(vars[bestj] >= IloFloor(x[bestj])+1,
                           vars[bestj] <= IloFloor(x[bestj])),
                    this);
   }

   x.end();
   obj.end();
   feas.end();
```

```
   return res;
}


int
main (int argc, char **argv)
{
   IloEnv   env;
   try {
      IloModel model(env);
      IloCplex cplex(env);

      if ( argc != 2 ) {
         usage (argv[0]);
         throw(-1);
      }

      IloObjective   obj;
      IloNumVarArray var(env);
      IloRangeArray  rng(env);
      cplex.importModel(model, argv[1], obj, var, rng);

      cplex.extract(model);
      cplex.solve(MyBranchGoal(env, var));

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      cout << "Solution status = " << cplex.getStatus() << endl;
      cout << "Solution value  = " << cplex.getObjValue() << endl;
      cout << "Values          = " << vals << endl;
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }

   env.end();

   return 0;
}


static void usage (const char *progname)
{
   cerr << "Usage: " << progname << " filename" << endl;
   cerr << "   where filename is a file with extension " << endl;
   cerr << "      MPS, SAV, or LP (lower case is allowed)" << endl;
   cerr << " Exiting..." << endl;
}
```

This example shows how to implement and use a goal for controlling the branch strategy used by `IloCplex`. As discussed, goals are implemented as subclasses of class `IloCplex::GoalI` (`IloCplex.Goal`). The C++ version of that example uses the macro

```
ILOCPLEXGOAL1(MyBranchGoal, IloNumVarArray, vars)
```

instead. This macro defines two things, class `MyBranchGoalI` and the function

```
IloCplex::Goal MyBranchGoal(IloEnv env, IloNumVarArray vars);
```

The class `MyBranchGoalI` is defined as a subclass of class `IloCplex::GoalI` (`IloCplex.Goal`) and has a private member `IloNumVarArray` vars. The function `MyBranchGoal` creates an instance of class `MyBranchGoalI`, initializes the member `vars` to the parameter `vars` passed to the function, and returns a handle to the new goal object. The curly brackets "{ ... }" following the macro enclose the implementation of method `MyBranchGoalI::execute` containing the actual code of the goal.

The use of the macro is very convenient as the amount of user code is equivalent to the amount for defining a function, but with a slightly unusual syntax. `IloCplex` provides 7 such macros that can be used for defining goals with 0 to 6 private members. If more than 6 members are needed, `IloCplex::GoalI` (`IloCplex.Goal`) must be subclassed *by hand*.

Since the Java programming language does not provide macros, a subclass of `IloCplex.Goal` must always be implemented by hand. In this example, this class is called `MyBranchGoal` and there is no helper function for creating an instance of that class (as the macro does in the case of C++).

The goal is then used for solving the extracted node by calling:

```
cplex.solve(MyBranchGoal(env, var));
```

for C++, or for Java:

```
cplex.solve(new MyBranchGoal(var));
```

instead of the usual `cplex.solve`. The rest of the `main` function contains nothing new and will not be discussed any further.

In the implementation of the goal, or more precisely its method `execute`, start by declaring and initializing some arrays. These arrays are then used by methods of class `IloCplex::GoalI` (`IloCplex.Goal`) to query information about the node subproblem and the solution of its relaxation. The method `getValues` is used to query the solution values for the variables in `vars`, method `getObjCoefs` is used to query the linear objective function coefficients for these variables, and method `getFeasibilities` is used to query feasibility statuses for them. The feasibility status of a variable indicates whether `IloCplex` considers the current solution value of the variable to be integer feasible or not. `IloCplex::GoalI` (`IloCplex.Goal`) provides a wealth of other query methods. For details, see the *ILOG CPLEX Reference Manual*.

Once you have gathered information about the variables, their objective coefficients, and their current feasibility statuses, compute the index of an integer infeasible variable in `vars`

that has the largest objective coefficients among the variables with largest integer infeasibility. That index is recorded in variable `bestj`.

Then create a new goal handle object `res`. By default, this is initialized to an empty goal. However, if an integer infeasible variable was found among those in `vars`, then variable `bestj` will be ≥ 0 and a nonempty goal will be assigned to `res`:

```
res = AndGoal(OrGoal(vars[bestj] >= IloFloor(x[bestj])+1,
                     vars[bestj] <= IloFloor(x[bestj])),
                     this);
```

This goal creates two branches, one for $\texttt{vars[bestj]} \leq \lfloor \texttt{x[bestj]} \rfloor$ and one for $\texttt{vars[bestj]} \geq \lfloor \texttt{x[bestj]} \rfloor + 1$ and continues branching in both subtrees with the same goal `this`. Finally, call method `end` for all temporary arrays and return goal `res`.

Since Java objects are garbage collected, there is no need for variable `res`. Instead, depending on the availability of an integer infeasible variable, the `null` goal is returned or the returned goal is created in the return statement itself:

```
return cplex.and(cplex.or(cplex.geGoal(_vars[bestj],
Math.floor(x[bestj]))+1,
                          cplex.leGoal(_vars[bestj],
Math.floor(x[bestj]))),
                 this);
```

The Goal Stack

To better understand how goals are executed, consider the concept of the *goal stack*. Every node has its own goal stack. When calling `cplex.solve(goal)`, the goal stack of the root node is simply initialized with `goal` and then calls the regular `cplex.solve` method.

When `IloCplex` processes a node, it pops the first goal from the node's goal stack and calls method `execute`. If a nonempty goal is returned, it is simply pushed back on the stack. `IloCplex` keeps doing this until the node becomes inactive or the node's goal stack becomes empty. When the node stack is empty, `IloCplex` continues with its built-in search strategy for the subtree rooted at this node.

In light of the goal stack, here are the different types of goals:

1. As already discussed, the `Or` goal creates child nodes. `IloCplex` first initializes the goal stack of every child node with a copy of the remaining goal stack of the current node. Then it pushes the goal passed as the parameter to the `Or` goal on the goal stack of the corresponding node. Finally, the current node is deactivated, and `IloCplex` continues search by picking a new active node from the tree to process.

2. The And goal simply pushes the goals passed as parameters onto the goal stack in reverse order. Thus, when popping the goals from the stack for execution, they will be executed in the same order as they were passed as parameters to method And goal.
3. When a Fail goal executes, the current node is simply deactivated, and IloCplex continues on another active node from the tree. In other words, IloCplex discontinues its search below the current node.
4. When a local cut goal is executed, its constraints are added to the node problem as local cuts and the relaxation is re-solved.
5. An empty goal cannot be executed. Thus, empty goals are not pushed on the goal stack. If the goal stack is empty, IloCplex continues with the built-in branching strategy.

With this understanding, consider further what really goes on when a goal returns

```
return AndGoal(OrGoal(var <= IloFloor(val), var >= IloFloor(val)+1), this);
```

The And goal is pushed onto the current node's goal stack, only to be immediately popped back off of it. When it is executed, it will push this on the goal stack and then the Or goal. Thus, the Or goal is the next goal that IloCplex pops and executes. The Or goal creates two subnodes, and initializes their goal stacks with copies of the goal stack of the current node. At this point both subnodes will have this on top of their goal stacks. Next, the Or goal will push a local cut goal for $\texttt{var} \leq \lfloor \texttt{val} \rfloor$ (where $\lfloor \texttt{val} \rfloor$ denotes the floor of val) on the goal stack of the first subnode. Similarly, it pushes a local cut goal for $\texttt{var} \geq \lfloor \texttt{val} \rfloor + 1$ on the goal stack of the second subnode. Finally, the current node is deactivated and IloCplex continues its search with a new active node from the tree.

When IloCplex processes one of the subnodes that have been created by the Or goal, it will pop and execute the first goal from the node's goal stack. As you just saw, this will be a local cut goal. Thus IloCplex adds the constraint to the node problem and re-solves the relaxation. Next, this will be popped from the goal stack and executed. This means that the same search strategy as implemented in the original goal is applied at that node.

Memory Management and Goals

Java uses garbage collection to handle all memory management issues. Thus the following only applies to the C++ library. Java users may safely skip ahead to *Cuts and Goals*.

To conserve memory, IloCplex only stores active nodes of the tree and deletes nodes as soon as they become inactive. When deleting nodes, IloCplex also deletes the goal stacks associated with them, including all goals they may still contain. In other words, IloCplex takes over memory management for goals.

It does so by keeping track of how many references to every goal are in use. As soon as this number drops to zero (0), the goal is automatically deleted. This technique is known as reference counting.

`IloCplex` implements reference counting in the handle class `IloCplex::Goal`. Every `IloCplex::GoalI` object maintains a count of how many `IloCplex::Goal` handle objects refer to it. The assignment operator, the constructors, and the destructor of class `IloCplex::Goal` are implemented in such a way as to keep the reference count up-to-date. This means that users should always access goals through handle objects, rather than keeping their own pointers to implementation objects.

Other than that, nothing special needs to be observed when dealing with goals. In particular, goals don't have `end` methods like other handle classes in ILOG Concert Technology. Instead, `IloCplex` goal objects are automatically deleted when no more references to them exist.

Local cut goals contain `IloRange` objects. Since the `IloRange` object is only applied when the goal is executed, method `end` must not be called for a range constraint from which a local cut goal is built. The goal will take over memory management for the constraints and call method `end` when the goal itself is destroyed. Also, an `IloRange` object can only be used in exactly one local cut goal. Similarly, method `end` must not be called for `IloRangeArray` objects that are passed to local cut goals. Also such arrays must not contain duplicate elements.

Going back to example `ilogoalex1.cpp`, you see that the method `end` is called for the temporary arrays `x`, `obj`, and `feas` at the end of the `execute` method. Though a bit hidden, two `IloRange` constraints are constructed for the goal, corresponding to the parameters of the `Or` goal. `IloCplex` takes over memory management for these two constraints as soon as they are enclosed in a goal. This takeover happens via the implicit constructor `IloCplex::Goal::Goal(IloRange rng)` that is called when the range constraints are passed as parameters to the `Or` goal.

In summary, the user is responsible for calling `end` on all ILOG Concert Technology objects created in a goal, except when they have been passed as parameters to a new goal.

Also, user code in the `execute` method is not allowed to modify existing ILOG Concert Technology objects in any way. `IloCplex` uses an optimized memory management system within goals for dealing with temporary objects. However, this memory management system cannot be mixed with the default memory management system used by ILOG Concert Technology. Thus, for example, it is illegal to add an element to array `vars` in the example, since this array has been created outside of the goal.

Cuts and Goals

Goals can also be used to add global cuts. Whereas local cuts are respected only in a subtree, global cuts are added to the entire problem and are therefore respected at every node after they have been added.

Global cuts can be added similarly to local cuts by using a global cut goal. A global cut goal is created with method `IloCplex::GoalI::GlobalCutGoal` (`IloCplex.globalCutGoal`). This method takes an `IloRange` or an `IloRangeArray` (`IloRange[]`) object as its parameter and returns a goal. When the goal executes, it adds the constraints as global cuts to the problem.

Example `ilogoalex2.cpp` shows the use of `IloCplex::GoalI::GlobalCutGoal` for solving the noswot MILP model. This is a relatively small model from the MIPLIB 3.0 test set, consisting only of 128 variables. Nonetheless, it is very hard to solve without adding special cuts.

Although it is now solvable directly, the computation time is in the order of several hours on state-of-the-art computers. However, cuts can be derived, and the addition of these cuts makes the problem solvable in a matter of minutes or seconds. These cuts are:

```
x21 - x22 <= 0
x22 - x23 <= 0
x23 - x24 <= 0
2.08*x11 + 2.98*x21 + 3.47*x31 + 2.24*x41 + 2.08*x51 +
0.25*w11 + 0.25*w21 + 0.25*w31 + 0.25*w41 + 0.25*w51 <= 20.25
2.08*x12 + 2.98*x22 + 3.47*x32 + 2.24*x42 + 2.08*x52 +
0.25*w12 + 0.25*w22 + 0.25*w32 + 0.25*w42 + 0.25*w52 <= 20.25
2.08*x13 + 2.98*x23 + 3.47*x33 + 2.24*x43 + 2.08*x53 +
0.25*w13 + 0.25*w23 + 0.25*w33 + 0.25*w43 + 0.25*w53 <= 20.25
2.08*x14 + 2.98*x24 + 3.47*x34 + 2.24*x44 + 2.08*x54 +
0.25*w14 + 0.25*w24 + 0.25*w34 + 0.25*w44 + 0.25*w54 <= 20.25
2.08*x15 + 2.98*x25 + 3.47*x35 + 2.24*x45 + 2.08*x55 +
0.25*w15 + 0.25*w25 + 0.25*w35 + 0.25*w45 + 0.25*w55 <= 16.25
```

These cuts have been derived after interpreting the model as a resource allocation model on five machines with scheduling, horizon constraints, and transaction times. The first three cuts break symmetries among the machines, while the others capture minimum bounds on transaction costs.

Of course the best way to solve the `noswot` model with these cuts is to simply add them to the model before calling the optimizer. However, for demonstration purposes, the cuts are added by means of a goal. The source code of this example can be found in the `examples/src` directory of the ILOG CPLEX distribution. The equivalent Java version appears as `GoalEx2.java` in the same location.

```
// ------------------------------------------------------------ -*- C++ -*-
// File: examples/src/ilogoalex2.cpp
// Version 9.0
// ---------------------------------------------------------------------------
```

```
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ---------------------------------------------------------------------------
//
// ilogoalex2.cpp -- Solve noswot by adding cuts through the goal API
//

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

ILOCPLEXGOAL3(CutGoal, IloExprArray, lhs, IloNumArray, rhs, IloNum, eps) {
   if ( isIntegerFeasible() )
      return 0;

   IloCplex::Goal goal = this;
   IloInt n = lhs.getSize();
   IloBool foundCut = IloFalse;
   for (IloInt i = 0; i < n; ++i) {
      IloNum xrhs = rhs[i];
      if ( xrhs < IloInfinity  &&  getValue(lhs[i]) > xrhs + eps ) {
         goal = AndGoal(GlobalCutGoal(lhs[i] <= xrhs), goal);
         rhs[i] = IloInfinity;
         foundCut = IloTrue;
      }

      if ( !foundCut ) {
         goal = AndGoal(BranchAsCplexGoal(getEnv()), goal);
      }
   }
   return goal;
}

void
makeCuts(const IloNumVarArray vars, IloExprArray lhs, IloNumArray rhs)
{
   IloNumVar x11, x12, x13, x14, x15;
   IloNumVar x21, x22, x23, x24, x25;
   IloNumVar x31, x32, x33, x34, x35;
   IloNumVar x41, x42, x43, x44, x45;
   IloNumVar x51, x52, x53, x54, x55;
   IloNumVar w11, w12, w13, w14, w15;
   IloNumVar w21, w22, w23, w24, w25;
   IloNumVar w31, w32, w33, w34, w35;
   IloNumVar w41, w42, w43, w44, w45;
   IloNumVar w51, w52, w53, w54, w55;
   IloInt num = vars.getSize();

   for (IloInt i = 0; i < num; ++i) {
      if      ( strcmp(vars[i].getName(), "X11") == 0 ) x11 = vars[i];
      else if ( strcmp(vars[i].getName(), "X12") == 0 ) x12 = vars[i];
```

```
   else if ( strcmp(vars[i].getName(), "X13") == 0 ) x13 = vars[i];
   else if ( strcmp(vars[i].getName(), "X14") == 0 ) x14 = vars[i];
   else if ( strcmp(vars[i].getName(), "X15") == 0 ) x15 = vars[i];
   else if ( strcmp(vars[i].getName(), "X21") == 0 ) x21 = vars[i];
   else if ( strcmp(vars[i].getName(), "X22") == 0 ) x22 = vars[i];
   else if ( strcmp(vars[i].getName(), "X23") == 0 ) x23 = vars[i];
   else if ( strcmp(vars[i].getName(), "X24") == 0 ) x24 = vars[i];
   else if ( strcmp(vars[i].getName(), "X25") == 0 ) x25 = vars[i];
   else if ( strcmp(vars[i].getName(), "X31") == 0 ) x31 = vars[i];
   else if ( strcmp(vars[i].getName(), "X32") == 0 ) x32 = vars[i];
   else if ( strcmp(vars[i].getName(), "X33") == 0 ) x33 = vars[i];
   else if ( strcmp(vars[i].getName(), "X34") == 0 ) x34 = vars[i];
   else if ( strcmp(vars[i].getName(), "X35") == 0 ) x35 = vars[i];
   else if ( strcmp(vars[i].getName(), "X41") == 0 ) x41 = vars[i];
   else if ( strcmp(vars[i].getName(), "X42") == 0 ) x42 = vars[i];
   else if ( strcmp(vars[i].getName(), "X43") == 0 ) x43 = vars[i];
   else if ( strcmp(vars[i].getName(), "X44") == 0 ) x44 = vars[i];
   else if ( strcmp(vars[i].getName(), "X45") == 0 ) x45 = vars[i];
   else if ( strcmp(vars[i].getName(), "X51") == 0 ) x51 = vars[i];
   else if ( strcmp(vars[i].getName(), "X52") == 0 ) x52 = vars[i];
   else if ( strcmp(vars[i].getName(), "X53") == 0 ) x53 = vars[i];
   else if ( strcmp(vars[i].getName(), "X54") == 0 ) x54 = vars[i];
   else if ( strcmp(vars[i].getName(), "X55") == 0 ) x55 = vars[i];
   else if ( strcmp(vars[i].getName(), "W11") == 0 ) w11 = vars[i];
   else if ( strcmp(vars[i].getName(), "W12") == 0 ) w12 = vars[i];
   else if ( strcmp(vars[i].getName(), "W13") == 0 ) w13 = vars[i];
   else if ( strcmp(vars[i].getName(), "W14") == 0 ) w14 = vars[i];
   else if ( strcmp(vars[i].getName(), "W15") == 0 ) w15 = vars[i];
   else if ( strcmp(vars[i].getName(), "W21") == 0 ) w21 = vars[i];
   else if ( strcmp(vars[i].getName(), "W22") == 0 ) w22 = vars[i];
   else if ( strcmp(vars[i].getName(), "W23") == 0 ) w23 = vars[i];
   else if ( strcmp(vars[i].getName(), "W24") == 0 ) w24 = vars[i];
   else if ( strcmp(vars[i].getName(), "W25") == 0 ) w25 = vars[i];
   else if ( strcmp(vars[i].getName(), "W31") == 0 ) w31 = vars[i];
   else if ( strcmp(vars[i].getName(), "W32") == 0 ) w32 = vars[i];
   else if ( strcmp(vars[i].getName(), "W33") == 0 ) w33 = vars[i];
   else if ( strcmp(vars[i].getName(), "W34") == 0 ) w34 = vars[i];
   else if ( strcmp(vars[i].getName(), "W35") == 0 ) w35 = vars[i];
   else if ( strcmp(vars[i].getName(), "W41") == 0 ) w41 = vars[i];
   else if ( strcmp(vars[i].getName(), "W42") == 0 ) w42 = vars[i];
   else if ( strcmp(vars[i].getName(), "W43") == 0 ) w43 = vars[i];
   else if ( strcmp(vars[i].getName(), "W44") == 0 ) w44 = vars[i];
   else if ( strcmp(vars[i].getName(), "W45") == 0 ) w45 = vars[i];
   else if ( strcmp(vars[i].getName(), "W51") == 0 ) w51 = vars[i];
   else if ( strcmp(vars[i].getName(), "W52") == 0 ) w52 = vars[i];
   else if ( strcmp(vars[i].getName(), "W53") == 0 ) w53 = vars[i];
   else if ( strcmp(vars[i].getName(), "W54") == 0 ) w54 = vars[i];
   else if ( strcmp(vars[i].getName(), "W55") == 0 ) w55 = vars[i];
}

lhs.add(x21 - x22);
rhs.add(0.0);
```

```
   lhs.add(x22 - x23);
   rhs.add(0.0);

   lhs.add(x23 - x24);
   rhs.add(0.0);

   lhs.add(2.08*x11 + 2.98*x21 + 3.47*x31 + 2.24*x41 + 2.08*x51 +
           0.25*w11 + 0.25*w21 + 0.25*w31 + 0.25*w41 + 0.25*w51);
   rhs.add(20.25);

   lhs.add(2.08*x12 + 2.98*x22 + 3.47*x32 + 2.24*x42 + 2.08*x52 +
           0.25*w12 + 0.25*w22 + 0.25*w32 + 0.25*w42 + 0.25*w52);
   rhs.add(20.25);

   lhs.add(2.08*x13 + 2.98*x23 + 3.47*x33 + 2.24*x43 + 2.08*x53 +
           0.25*w13 + 0.25*w23 + 0.25*w33 + 0.25*w43 + 0.25*w53);
   rhs.add(20.25);

   lhs.add(2.08*x14 + 2.98*x24 + 3.47*x34 + 2.24*x44 + 2.08*x54 +
           0.25*w14 + 0.25*w24 + 0.25*w34 + 0.25*w44 + 0.25*w54);
   rhs.add(20.25);

   lhs.add(2.08*x15 + 2.98*x25 + 3.47*x35 + 2.24*x45 + 2.08*x55 +
           0.25*w15 + 0.25*w25 + 0.25*w35 + 0.25*w45 + 0.25*w55);
   rhs.add(16.25);
}


int
main(int argc, char** argv)
{
   IloEnv env;
   try {
      IloModel m;
      IloCplex cplex(env);

      IloObjective   obj;
      IloNumVarArray var(env);
      IloRangeArray  con(env);

      env.out() << "reading ../../../examples/data/noswot.mps" << endl;
      cplex.importModel(m, "../../../examples/data/noswot.mps", obj, var, con);

      env.out() << "extracting model ..." << endl;
      cplex.extract(m);
      cplex.setParam(IloCplex::MIPInterval, 1000);

      env.out() << "initializing cuts ..." << endl;
      IloExprArray lhs(env);
      IloNumArray  rhs(env);
      makeCuts(var, lhs, rhs);
```

```
      env.out() << "solving model ...\n";
      cplex.solve(CutGoal(env, lhs, rhs, cplex.getParam(IloCplex::EpRHS)));
      env.out() << "solution status is " << cplex.getStatus() << endl;
      env.out() << "solution value  is " << cplex.getObjValue() << endl;
   }
   catch (IloException& ex) {
      cerr << "Error: " << ex << endl;
   }
   env.end();
   return 0;
}
```

The goal `CutGoal` in this example receives a list of "less than" constraints to use as global cuts and a tolerance value `eps`. The constraints are passed to the goal as an array of `lhs` expressions and an array of corresponding `rhs` values. Both are initialized in function `makeCuts`.

The goal `CutGoal` checks whether any of the constraints passed to it are violated by more than the tolerance value. It adds violated constraints as global cuts. Other than that, it follows the branching strategy `IloCplex` would use on its own.

The goal starts out by checking if the solution of the continuous relaxation of the current node subproblem is integer feasible. This is done by calling method `isIntegerFeasible`. If the current solution is integer feasible, a candidate for a new incumbent has been found and the goal returns the empty goal to instruct `IloCplex` to continue on its own.

Otherwise, the goal checks if any of the constraints passed to it are violated. It computes the value of every `lhs` expression for current solution by calling `getValue(lhs[i])`. The result is compared against the corresponding `rhs` value `rhs[i]`. If a violation of more than `eps` is detected, the constraint is added as a global cut and the `rhs` value will be set to `IloInfinity` to avoid checking it again unnecessarily.

The global cut goal for `lhs[i]` $\leq$ `rhs[i]` is created by calling method `GlobalCutGoal`. It is then combined with goal `goal` using method `AndGoal`, so that the new global cut goal will be executed first. The resulting goal is stored again in variable `goal`. Before adding any global cut goals, variable `goal` is initialized as

```
IloCplex::Goal goal = AndGoal(BranchAsCplexGoal(getEnv()), this);
```

for C++, or for Java:

```
cplex.and(cplex.branchAsCplex(), this);
```

The method `BranchAsCplexGoal(getEnv)` (`cplex.branchAsCplex`) creates a goal that branches in the same way as the built-in branch procedure. By adding this goal, the current goal will be executed for the entire subtree.

Thus the goal returned by `CutGoal` will add all currently violated constraints as global cuts one by one. Then it will branch in the way `IloCplex` would branch without any goals and execute the `CutGoal` again in the child nodes.

Injecting Heuristic Solutions

At any time in the execution of a goal, you may find that, for example, by slightly manipulating the current node subproblem solution, you may construct a solution to your model. Such solutions are called *heuristic solutions*, and a procedure that generates them is called a *heuristic*.

Heuristic solutions can be injected into the branch & cut search by creating a solution goal with method `IloCplex::GoalI::SolutionGoal` (`IloCplex.solutionGoal`). Such a goal can be returned typically as a subgoal of `And` goal much like global cut goals.

When `IloCplex` executes a solution goal it does not immediately use the specified solution as a potential new incumbent. The reason is that with goals, part of the model may be specified via global cuts or through specialized branching strategies. Thus the solution needs first to be tested for feasibility with respect to the entire model including any part of the model specified through goals.

To test if an injected solution is feasible with all goals the `execute` method of a solution goal `IloCplex` first creates a subnode of the current node. This subnode will of course inherit the goal stack from its parent. In addition the solution goal will push local cuts onto the stack of the subnode such that all variables are fixed to the values of the injected solution.

By processing this subnode (as the next node), `IloCplex` makes sure that the solution is feasible with respect to all goals, or is otherwise discarded. Goals that have been executed so far are either reflected as global cuts or by the local cuts that are active at the current node. Thus, if the relaxation remains feasible after the variable fixings have been added, the feasibility of these goals is certain.

If at that point the goal stack is not empty, the goals on the goal stack need to be checked for feasibility as well. Thus by continuing to execute the goals from the goal stack `IloCplex` will either prove feasibility of the solution with respect to the remaining goals or, in case the relaxation becomes infeasible, determine it to be really infeasible and discard the solution. The rest of the branch & cut search remain unaffected by all of this.

The benefit of this approach is that your heuristic need not be aware of the entire model including all its parts that might be implemented via goals. Your heuristic can still safely be used, as `IloCplex` will make sure of feasibility for the entire model. However, there are some performance considerations to observe. If parts of the model specified with goals are dominant, heuristic solutions you generate might need to be rejected so frequently that you do not get enough payoff for the work of running the heuristic. Also, your heuristic should

account for the global and local cuts that have been added at the node where you run your heuristic so that a solution candidate is not rejected right away and the work wasted.

Controlling Goal-Defined Search

So far, you have seen how to control the branching and cut generation of `IloCplex` branch & cut search. The remaining missing piece is the node selection strategy. The node selection strategy determines which of the active nodes in the tree `IloCplex` chooses when it selects the next node for processing. `IloCplex` has several built-in node selection strategies, selected through parameter `NodeSel`.

When using goal-controlled search, node evaluators are used to override the built-in node selection strategy. The process is as follows. You combine a goal with a node evaluator by calling method `IloCplex::Goal::Apply` (`IloCplex.apply`). This method returns a new goal that implements the same search strategy as the goal passed as the parameter, but adds the node evaluator to every node in the subtree defined by the goal. Therefore, nodes may have a list of evaluators attached to them.

When node evaluators are used, nodes are selected like this:

- `IloCplex` starts to choose the node with the built-in strategy as a first candidate.
- Then `IloCplex` loops over all remaining active nodes and considers choosing them instead.
- If a node has the same evaluator attached to it as the current candidate, the evaluator is asked if this node should take precedence over the current candidate. If the response is positive, the node under investigation becomes the new candidate and the test against other nodes continues.

If a node has multiple evaluators attached, they are consulted in the order the evaluators have been applied. This occurs as follows:

- If the first evaluator prefers one node over the other, the preferred node is used as candidate and the next node is considered.
- If the first evaluator does not give preference to one node over the other, the second evaluator is considered, and so on.

Thus, by adding multiple evaluators, you can build composite node selection strategies where later evaluators are used for breaking ties in previous ones.

Node evaluators are implemented as subclasses of class `IloCplex::NodeEvaluatorI`. Class `IloCplex::NodeEvaluator` is the handle class for node evaluators. In Java, node evaluators are implemented in objects of type `IloCplex.NodeEvaluator` (and there are no handle classes). Like goals, node evaluators use reference counting for memory management. As a result, you should always use the handle objects when dealing with node evaluators, and there is no method `end` to be called.

Node evaluators use a two-step process to decide whether one node should take precedence over another. First, the evaluator computes a value for every node to which it is attached. This is done by calling method

```
IloNum IloCplex::NodeEvaluatorI::evaluate();
```

in the case of C++, and for Java by calling method:

```
double IloCplex.NodeEvaluator.evaluate();
```

This method must be implemented by users who write their own node evaluators. In the method `evaluate`, the protected methods of the class `IloCplex::NodeEvaluatorI` (`IloCplex.NodeEvaluator`) can be called to query information about the node being evaluated. The method `evaluate` must compute and return an evaluation (that is, a value) that is used later on, in the second step, to compare two nodes and select one of them. The `evaluate` method is called only once for every node, and the result is cached and reused whenever the node is compared against another node with the evaluator.

The second step consists of comparing the current candidate to another node. This only happens for evaluators that are shared by the current candidate and the other node. By default, the candidate is replaced by the other node if its evaluation value is smaller than that of the candidate. This behavior can be altered by overwriting the method

```
IloBool IloCplex::NodeEvaluatorI::subsume(IloNum candVal, IloNum nodeVal);
```

or, in the case of Java:

```
boolean IloCplex.NodeEvaluator.subsume(double candVal, double nodeVal);
```

`IloCplex` calls this method of an evaluator attached to the current candidate if the node being compared also has the same evaluator attached. The first parameter `candVal` is the evaluation value the evaluator has previously computed for the current candidate, and `nodeVal` is the evaluation value the evaluator has previously computed for the node being tested. If this method returns `IloTrue` (`true`), the candidate is replaced. Otherwise the method is called again with reversed parameters. If it still returns `IloFalse` (`false`), both nodes are tied with respect to that evaluator, and the next evaluator they share is consulted. Otherwise, the current candidate is kept and tested against the next node.

There are two more virtual methods defined for node evaluators that should be considered when implementing your own node evaluator. The method `init` is called right before `evaluate` is called for the first time, thus allowing you to initialize internal data of the evaluator. When this happens, the evaluator has been initialized to the first node to be evaluated, thus information about this node can be queried by calling the methods of class `IloCplex::NodeEvaluatorI` (`IloCplex.NodeEvaluator`).

Finally, for C++ method

```
IloCplex::NodeEvaluatorI* IloCplex::NodeEvaluatorI::duplicateEvaluator();
```

must be implemented by the user to return a copy of the invoking node evaluator object. This method is called by `IloCplex` to create copies of the evaluator for parallel branch & cut search.

Example `ilogoalex3.cpp` shows how to use node evaluators to implement a node selection strategy that chooses the deepest active node in tree among those with maximal sum of integer infeasibilities. Example `ilogoalex3.cpp` can be found in the `examples/src` directory of your distribution. The equivalent Java version can be found as file `Goalex3.java` at the same location.

```
// ------------------------------------------------------------- -*- C++ -*-
// File: examples/src/ilogoalex3.cpp
// Version 9.0
// ------------------------------------------------------------------------
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ------------------------------------------------------------------------
//
// ilogoalex3.cpp - Node selectors in goal search
//
//                  This is an extension of example ilogoalex1.cpp
//                  It adds node evaluators to further control the
//                  goal based search.  The node evaluator chooses
//                  the node that is deepest in the tree among those
//                  with maximum sum of integer infeasibilities
//

#include <ilcplex/ilocplex.h>

ILOSTLBEGIN

static void usage (const char *progname);

// Branch on var with largest objective coefficient
// among those with largest infeasibility

ILOCPLEXGOAL1(MyBranchGoal, IloNumVarArray, vars) {
   IloNumArray x;
   IloNumArray obj;
   IntegerFeasibilityArray feas;

   x    = IloNumArray(getEnv());
   obj  = IloNumArray(getEnv());
   feas = IntegerFeasibilityArray(getEnv());
   getValues(x, vars);
   getObjCoefs(obj, vars);
   getFeasibilities(feas, vars);

   IloInt bestj  = -1;
   IloNum maxinf = 0.0;
```

```
   IloNum maxobj = 0.0;
   IloInt cols = vars.getSize();
   for (IloInt j = 0; j < cols; ++j) {
      if ( feas[j] == Infeasible ) {
         IloNum xj_inf = x[j] - IloFloor (x[j]);
         if ( xj_inf > 0.5 )  xj_inf = 1.0 - xj_inf;
         if ( xj_inf >= maxinf                              &&
             (xj_inf > maxinf || IloAbs (obj[j]) >= maxobj)  ) {
            bestj  = j;
            maxinf = xj_inf;
            maxobj = IloAbs (obj[j]);
         }
      }
   }

   IloCplex::Goal res;
   if ( bestj >= 0 ) {
      res = AndGoal(OrGoal(vars[bestj] >= IloFloor(x[bestj])+1,
                           vars[bestj] <= IloFloor(x[bestj])),
                    this);
   }

   x.end();
   obj.end();
   feas.end();

   return res;
}


// Depth first search node evaluator

class DepthEvaluatorI : public IloCplex::NodeEvaluatorI {
public:
   IloNum evaluate() const {
      return -getDepth();
   }

   IloCplex::NodeEvaluatorI *duplicateEvaluator() {
      return new DepthEvaluatorI();
   }
};

IloCplex::NodeEvaluator DepthEvaluator() {
   return new DepthEvaluatorI();
}


// integer infeasibility sum node evaluator

class IISumEvaluatorI : public IloCplex::NodeEvaluatorI {
public:
```

```
   IloNum evaluate() const {
      return -getInfeasibilitySum();
   }

   IloCplex::NodeEvaluatorI *duplicateEvaluator() {
      return new IISumEvaluatorI();
   }
};

IloCplex::NodeEvaluator IISumEvaluator() {
   return new IISumEvaluatorI();
}


int
main (int argc, char **argv)
{
   IloEnv env;
   try {
      IloModel model(env);
      IloCplex cplex(env);

      if ( argc != 2 ) {
         usage (argv[0]);
         throw(-1);
      }

      IloObjective   obj;
      IloNumVarArray var(env);
      IloRangeArray  rng(env);
      cplex.importModel(model, argv[1], obj, var, rng);

      cplex.extract(model);

      IloCplex::Goal iiSumGoal = IloCplex::Apply(cplex,
                                                 MyBranchGoal(env, var),
                                                 IISumEvaluator());
      IloCplex::Goal depthGoal = IloCplex::Apply(cplex,
                                                 iiSumGoal,
                                                 DepthEvaluator());
      cplex.solve(depthGoal);

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      cout << "Solution status = " << cplex.getStatus() << endl;
      cout << "Solution value  = " << cplex.getObjValue() << endl;
      cout << "Values          = " << vals << endl;
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }

   env.end();
```

```
   return 0;
}

static void usage (const char *progname)
{
   cerr << "Usage: " << progname << " filename" << endl;
   cerr << "   where filename is a file with extension " << endl;
   cerr << "      MPS, SAV, or LP (lower case is allowed)" << endl;
   cerr << " Exiting..." << endl;
}
```

As this example is an extension of example `ilogoalex1.cpp`, it concentrates only on the differences. Also, the example is discussed only in terms of the C++ version; the Java version has identical structure and design and differs only in syntax.

The first is the definition of class `DepthEvaluatorI` as a subclass of `IloCplex::NodeEvaluatorI`. It implement the methods `evaluate` and `duplicateEvaluator`. The method `evaluate` simply returns the negative depth value queried for the current node by calling method `getDepth`. Since `IloCplex` by default chooses nodes with the lowest evaluation value, this evaluator will favor nodes deep in the tree. The method `duplicateEvaluator` simply returns a copy of the invoking object by calling the (default) copy constructor. Along with the class, the function `DepthEvaluator` is also defined to create an instance of class `DepthEvaluatorI` and returns a handle to it.

Similarly, the class `IISumEvaluatorI` and function `IISumEvaluator` are also defined. The `evaluate` method returns the negation of the sum of integer infeasibilities of the node being evaluated. This number is obtained by calling method `getInfeasibilitySum`. Thus, this evaluator favors nodes with larger sums of integer infeasibilities.

This example uses the same search strategy as `ilogoalex1.cpp`, implemented in goal `MyBranchGoal`. However, it applies first the `IISumEvaluator` to select nodes with high integer infeasibility sum, to choose between nodes with the same integer infeasibility sum it applies the `DepthEvaluator`. Applying the `IISumEvaluator` is done with

```
IloCplex::Goal iiSumGoal = IloCplex::Apply(cplex,
                                           MyBranchGoal(env, var),
                                           IISumEvaluator());
```

The goal created by calling `MyBranchGoal` is merged with the evaluator created by calling `IISumEvaluator` into a new goal `iiSumGoal`. Similarly, the `iiSumGoal` is merged with the node evaluator created by calling `DepthEvaluator` into a new goal `depthGoal`:

```
IloCplex::Goal depthGoal = IloCplex::Apply(cplex,
                                           iiSumGoal,
                                           DepthEvaluator());
```

Thus, `depthGoal` represents a goal implementing the branching strategy defined by `MyBranchGoal`, but using `IISumEvaluator` as a primary node selection strategy and

DepthEvaluator as a secondary node selection strategy for breaking ties. This goal is finally used for the branch & cut search by passing it to the solve method.

Node evaluators are only active while the search is controlled by goals. That is, if the goal stack becomes empty at a node and IloCplex continues searching with its built-in search strategy, that search is no longer controlled by any node evaluator. In order to maintain control over the node selection strategy while using the IloCplex branch strategy, you can use the goal returned by method IloCplex::GoalI::BranchAsCplexGoal (IloCplex.branchAsCplex). A goal that follows the branching performed by IloCplex built-in strategy can be easily implemented as:

```
ILOCPLEXGOAL0(DefaultSearchGoal) {
  if ( !isIntegerFeasible() )
    return AndGoal(BranchAsCplexGoal(getEnv()), this);
  return 0;
}
```

Notice the test for integer feasibility. Without that test, the application would create an endless loop because when an integer feasible solution has been found, BranchAsCplex goal does not change the node at all, and this would continue to be executed indefinitely.

Search Limits

As with node evaluators, it is possible to apply search limits to the branch & cut search controlled by goals. Search limits allow you to limit the search in certain subtrees; that is, they allow you to discontinue processing nodes when some condition applies. Search limits are implemented in subclasses of class IloCplex::SearchLimitI (IloCplex.SearchLimit), and the procedure for implementing and using them is very similar to that for node evaluators. See the reference manual for more details on implementing and using search limits.

C H A P T E R

22

Using Callbacks

This chapter introduces callbacks. Callbacks allow you to monitor closely and to guide the behavior of ILOG CPLEX optimizers. In particular, ILOG CPLEX callbacks allow user code to be executed regularly during an optimization. To use callbacks with ILOG CPLEX, you must first write the callback function, and then pass it to ILOG CPLEX. There are two types of callbacks: diagnostic callbacks and control callbacks. You will find additional information about callbacks in this manual in *Advanced MIP Control Interface* on page 513. This chapter includes information about:

- *Diagnostic Callbacks* on page 478
- *Implementing Callbacks in ILOG CPLEX with Concert Technology* on page 478
- *Example: Deriving the Simplex Callback ilolpex4.cpp* on page 482
- *Implementing Callbacks in the Callable Library* on page 486
- *Interaction Between Callbacks and ILOG CPLEX Parallel Optimizers* on page 487
- *Example: Using Callbacks lpex4.c* on page 488
- *Control Callbacks for IloCplex* on page 496
- *Example: Controlling Cuts iloadmipex5.cpp* on page 497

Notes: *The callback class hierarchy for Java and C#.NET is exactly the same as the hierarchy for C++, but the class names differ, in that there is no* `I` *at the end.*

For example, the Java implementation class corresponding to the C++ class `IloCplex::ContinuousCallbackI` *is* `IloCplex.ContinuousCallback`.

The names of callback classes in C#.NET correspond very closely to those in the Java API. However, the name of a C#.NET class does not begin with Ilo. Furthermore, the names of C#.NET methods are capitalized (that is, they begin with an uppercase character) according to .NET conventions.

For example, the corresponding callback class in C#.NET is Cplex.ContinuousCallback.

Diagnostic Callbacks

Diagnostic callbacks allow you to monitor an ongoing optimization, and optionally to abort it. These callbacks are distinguished by the place where they are called during an optimization. There are nine such places where diagnostic callbacks are called:

- The presolve callback is called regularly during presolve.
- The crossover callback is called regularly during crossover from a barrier solution to a simplex basis.
- The network callback is called regularly during the network simplex.
- The barrier callback is called at each iteration during the barrier algorithm.
- The simplex callback is called at each iteration during the simplex algorithm.
- The MIP callback is called at each node during the branch & cut search.
- The probing callback is called regularly during probing.
- The fractional cut callback is called regularly during the generation of fractional cuts.
- The disjunctive cut callback is called regularly during the generation of disjunctive cuts.

Implementing Callbacks in ILOG CPLEX with Concert Technology

Callbacks are accessed via the `IloCplex::Callback` handle class in the C++ version of `IloCplex`. It points to an implementation object of a subclass of `IloCplex::CallbackI`. In Java and C#.NET, there is no handle class and a programmer deals only with implementation classes which are subclasses of `IloCplex.Callback`. One such

implementation class is provided for each type of callback. The implementation class provides the functions that can be used for the particular callback as protected methods.

To reflect the fact that some callbacks share part of their protected API, the callback classes are organized in a class hierarchy, as documented in the reference manuals of the APIs. For example, the class hierarchy of C++ callbacks is visible when you select Tree in the reference manual of that API. Likewise, the class and interface hierarchy of Java callbacks is visible when you select Tree in the reference manual of the Java API. Similarly, you can see the class and interface hierarchy of C#.NET callbacks in that reference manual.

This hierarchy means that, for example, all functions available for the MIP callback are also available for the probing, fractional cut, and disjunctive cut callbacks. In particular, the function to abort the current optimization is provided by the class `IloCplex::CallbackI` (`IloCplex.Callback`) and is thus available to all callbacks.

There are two ways of implementing callbacks for `IloCplex`: a more complex way that exposes all the C++ implementation details, and a simplified way that uses macros to handle the C++ technicalities. Since the Java programing language does not provide macros, only the more complex way is available for Java users. This section first explains the more complex way and discusses the underlying design. To quickly implement your callback without details about the internal design, proceed directly to *Writing Callbacks with Macros* on page 480.

Writing Callback Classes by Hand

To implement your own callback for `IloCplex`, first select the callback class corresponding to the callback you want implemented. From it derive your own implementation class and overwrite the virtual method `main`. This is where you implement the callback actions, using the protected methods of the callback class from which you derived your callback or one of its base classes.

Next write a function that creates a new object of your implementation class using the environment operator `new` and returning it as an `IloCplex::Callback` handle object. Here is an example implementation of such a function:

```
IloCplex::Callback MyCallback(IloEnv env, IloInt num) {
    return (new (env) MyCallbackI(num));
}
```

It is not customary to write such a function for Java, but `new` is called explicitly for creating a callback object when needed. Once an implementation object of your callback is created (either with the constructor function in C++ or by directly calling the `new` operator for Java), use it with `IloCplex` by calling `cplex.use` with the callback object as parameter. In C++, to remove a callback that is being used by a `cplex` object, call `callback.end` on the `IloCplex::Callback` handle callback. In java there is no way of removing individual callbacks from your `IloCplex` object. Instead, you can remove all callbacks by calling

cplex.clearCallbacks. Since Java uses garbage collection for memory management, there is nothing equivalent to the end method for callbacks in the Java API.

One object of a callback implementation class can be used with only one IloCplex object at a time. Thus, when you use a callback with more than one cplex object, a copy of the implementation object is created every time cplex.use is called except for the first time. In C++, the method IloCplex::use returns a handle to the callback object that has actually been installed to enable calling end on it.

To construct the copies of the callback objects in C++, class IloCplex::CallbackI defines another pure virtual method:

```
virtual IloCplex::CallbackI*
IloCplex::CallbackI::duplicateCallback()
     const = 0;
```

which must be implemented for your callback class. This method will be called to create the copies needed for using a callback on different cplex objects or on one cplex object with a parallel optimizer.

In most cases you can avoid writing callback classes by hand, using supplied macros that make the process as easy as implementing a function. You must implement a callback by hand only if the callback manages internal data not passed as arguments, or if the callback requires eight or more parameters.

Writing Callbacks with Macros

This is how to implement a callback using macros. Since macros are not supported in Java, this technique will only apply to C++ applications.

Start by determining which callback you want to implement and how many arguments to pass to the callback function. These two pieces of information determine the macro you need to use.

For example, to implement a simplex callback with one parameter, the macro is ILOSIMPLEXCALLBACK1. Generally, for every callback type XXX and any number of parameters n from 0 to 7, there is a macro called ILOXXXCALLBACKn. Table 22.1 lists the callbacks and the corresponding macros and classes (where n is a placeholder for 0 to 7).

Table 22.1 *Callback Macros*

Callback	Macro	Class
presolve	ILOPRESOLVECALLBACKn	IloCplex::PresolveCallbackI
continuous	ILOCONTINUOUSCALLBACKn	IloCplex::ContinuousCallbackI
simplex	ILOSIMPLEXCALLBACKn	IloCplex::SimplexCallbackI
barrier	ILOBARRIERCALLBACKn	IloCplex::BarrierCallbackI
crossover	ILOCROSSOVERCALLBACKn	IloCplex::CrossoverCallbackI
network	ILONETWORKCALLBACKn	IloCplex::NetworkCallbackI
MIP	ILOMIPCALLBACKn	IloCplex::MIPCallbackI
probing	ILOPROBINGCALLBACKn	IloCplex::ProbingCallbackI
fractional cut	ILOFRACTIONALCUTCALLBACKn	IloCplex::FractionalCutCallbackI
disjunctive cut	ILODISJUNCTIVECUTCALLBACKn	IloCplex::DisjunctiveCutCallbackI

The protected methods of the corresponding class and its base classes determine the functions that can be called for implementing your callback. See the *ILOG CPLEX Reference Manual*.

Here is an example of how to implement a simplex callback with the name `MyCallback` that takes one parameter:

```
ILOSIMPLEXCALLBACK1(MyCallback, IloInt, num) {
  if ( getNiterations() == num ) abort();
}
```

This callback aborts the simplex algorithm at the iteration indicated by the number `num`. It queries the current iteration number by calling the function `getNiterations`, a protected method of the class `IloCplex::ContinuousCallbackI`.

To use this callback with an `IloCplex` object `cplex`, simply call:

```
IloCplex::Callback mycallback = cplex.use(MyCallback(env, 10));
```

The callback that is added to `cplex` is returned by the method `use` and stored in variable `mycallback`. This allows you to call `mycallback.end`to remove the callback from `cplex`. If you do not intend accessing your callback, for example in order to delete it before ending the environment, you may safely leave out the declaration and initialization of variable `mycallback`.

Callback Interface

Two callback classes in the hierarchy need extra attention. The first is the base class `IloCplex::CallbackI` (`IloCplex.CallbackI`). Since there is no corresponding callback in the Callable Library, this class cannot be used for implementing user callbacks. Instead, its purpose is to provide an interface common to all callback functions. This consists of the methods `getModel`, which returns the model that is extracted to the `IloCplex` object that is calling the callback, `getEnv`, which returns the corresponding environment (C++ only), and `abort`, which aborts the current optimization. Further, methods `getNrows` and `getNcols` allow you to query the number of rows and columns of the current `cplex` LP matrix. These methods can be called from all callbacks.

> ***Note:*** *For C++ users, no manipulation of the model or, more precisely, any extracted modeling object is allowed during the execution of a callback. No modification is allowed of any array or expression not local to the callback function itself (that is, constructed and* `end`*ed in it). The only exception is the modification of array elements. For example,* `x[i] = 0` *would be permissible, whereas* `x.add(0)` *would not unless* `x` *is a local array of the callback.*

The Continuous Callback

The second special callback class is `IloCplex::ContinuousCallbackI` (`IloCplex.ContinuousCallback`). If you create a Continuous callback and use it with an `IloCplex` object, it will be used for both the barrier and the simplex callback. In other words, implementing and using one Continuous callback is equivalent to writing and using these two callbacks independently.

Example: Deriving the Simplex Callback ilolpex4.cpp

Example `ilolpex4.cpp` demonstrates the use of the simplex callback to print logging information at each iteration. It is a modification of example `ilolpex1.cpp`, so this discussion concentrates on the differences. The following code:

```
ILOSIMPLEXCALLBACK0(MyCallback) {
  cout << "Iteration " << getNiterations() << ": ";
  if ( isFeasible() ) {
    cout << "Objective = " << getObjValue() << endl;
  }
  else {
    cout << "Infeasibility measure = " << getInfeasibility() << endl;
  }
}
```

defines the callback `MyCallback` without parameters with the code enclosed in the outer `{}`. In Java the same callback is defined as:

```
static class MyCallback extends IloCplex.ContinuousCallback {
   public void main() throws IloException {
      System.out.print("Iteration " + getNiterations() + ": ");
      if ( isFeasible() )
         System.out.println("Objective = " + getObjValue());
      else
         System.out.println("Infeasibility measure = "
                            + getInfeasibility());
   }
}
```

The callback prints the iteration number. Then, depending on whether the current solution is feasible or not, it prints the objective value or infeasibility measure. The functions `getNiterations`, `isFeasible`, `getObjValue`, and `getInfeasibility` are methods provided in the callback's base class `IloCplex::ContinuousCallbackI` (`IloCplex.ContinuousCallback`). See the *ILOG CPLEX Reference Manual* for the complete list of methods provided for each callback class.

Here is how the macro `ILOSIMPLEXCALLBACK0` is expanded:

```
class MyCallbackI : public IloCplex::SimplexCallbackI {
  void main();
  IloCplex::CallbackI* duplicateCallback() const {
    return (new (getEnv()) MyCallbackI(*this));
  }
};
IloCplex::Callback MyCallback(IloEnv env) {
  return (IloCplex::Callback(new (env) MyCallbackI()));
}
void MyCallbackI::main() {
  cout << "Iteration " << getNiterations() << ": ";
  if ( isFeasible() ) {
    cout << "Objective = " << getObjValue() << endl;
  }
  else {
     cout << "Infeasibility measure = " << getInfeasibility() << endl;
  }
}
```

The `0` (zero) in the macro indicates that no parameters are passed to the constructor of the callback. For callbacks requiring up to 7 parameters, similar macros are defined where the `0` is replaced by the number of parameters, ranging from 1 through 7. For an example using the cut callback, see *Example: Controlling Cuts iloadmipex5.cpp* on page 497. If you need more than 7 parameters, you will need to derive your callback class yourself without the help of a macro.

After the callback `MyCallback` is defined, it can be used with the line:

```
cplex.use(MyCallback(env));
```

in C++ or

```
   cplex.use(new MyCallback());
```

in Java.

In the case of C++, function `MyCallback` creates an instance of the implementation class `MyCallbackI`. A handle to this implementation object is passed to `cplex` method `use`.

If your application defines more than one simplex callback object (possibly with different subclasses), only the last one passed to ILOG CPLEX with the `use` method is actually used during simplex. On the other hand, `IloCplex` can handle one callback for each callback class at the same time. For example, a simplex callback and a MIP callback can be used at the same time.

The complete program, `ilolpex4.cpp`, appears here and online in the standard distribution.

```
// ------------------------------------------------------------- -*- C++ -*-
// File: examples/src/ilolpex4.cpp
// Version 9.0
// ------------------------------------------------------------------------
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ------------------------------------------------------------------------
//
// ilolpex4.cpp - Illustrating the CPLEX callback functionality.
//
// This is a modification of ilolpex1.c, where we use a callback
// function to print the iteration info, rather than have CPLEX
// do it.

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

ILOSIMPLEXCALLBACK0(MyCallback) {
  cout << "Iteration " << getNiterations() << ": ";
  if ( isFeasible() ) {
     cout << "Objective = " << getObjValue() << endl;
  } else {
     cout << "Infeasibility measure = " << getInfeasibility() << endl;
  }
}


static void
   populatebycolumn (IloModel model, IloNumVarArray var, IloRangeArray rng);

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

```
   try {
      IloModel model(env, "example");

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

      IloCplex cplex(model);
      cplex.setOut(env.getNullStream());
      cplex.setParam(IloCplex::RootAlg, IloCplex::Primal);
      cplex.use(MyCallback(env));
      cplex.solve();

      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;

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

      cplex.exportModel("lpex4.lp");
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

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


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

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

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

   x.add(IloNumVar(obj(1.0) + c[0](-1.0) + c[1]( 1.0), 0.0, 40.0));
```

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

   model.add(obj);
   model.add(c);
}  // END populatebycolumn
```

Implementing Callbacks in the Callable Library

ILOG CPLEX optimization routines in the Callable Library incorporate a callback facility to allow your application to transfer control temporarily from ILOG CPLEX to the calling application. Using callbacks, your application can implement interrupt capability, for example, or create displays of optimization progress. Once control is transferred back to a function in the calling application, the calling application can retrieve specific information about the current optimization from the routine `CPXgetcallbackinfo`. Optionally, the calling application can then tell ILOG CPLEX to discontinue optimization.

To implement and use a callback in your application, you must first write the callback function and then tell ILOG CPLEX about it. For more information about the ILOG CPLEX Callable Library routines for callbacks, see the *ILOG CPLEX Reference Manual*. In that reference manual, the group `optim.cplex.callable.callbacks` gives you direct access to all such routines.

Setting Callbacks

In the Callable Library, diagnostic callbacks are organized into two groups: LP callbacks and MIP callbacks. For each group, one callback function can be set by the functions `CPXsetlpcallbackfunc` and `CPXsetmipcallbackfunc`, respectively. You can distinguish between the actual callbacks by querying the parameter `wherefrom` passed to the callback function as a parameter by ILOG CPLEX.

The LP callback is also called during the solution of problems of type QP and QCP.

Callbacks for LP/QP/QCPs and for MIPs

ILOG CPLEX will evaluate two user-defined callback functions, one during the solution of LP, QP, and QCP problems and one during the solution of MIP problems. ILOG CPLEX calls the LP/QP/QCP callback once per iteration during the solution of an LP, QP, or QCP problem and periodically during the presolve. ILOG CPLEX calls the MIP callback periodically during the probing phase of MIP preprocessing, periodically during cut generation, and once before each subproblem is solved in the branch & cut process.

Every user-defined callback must have these arguments:

- `env`, a pointer to the ILOG CPLEX environment;
- `cbdata`, a pointer to ILOG CPLEX internal data structures needed by `CPXgetcallbackinfo`;
- `wherefrom`, indicates which optimizer is calling the callback;
- `cbhandle`, a pointer supplied when your application calls `CPXsetlpcallbackfunc` or `CPXsetmipcallbackfunc` (so that the callback has access to private user data).

The arguments `wherefrom` and `cbdata` should be used only in calls to `CPXgetcallbackinfo`.

Return Values for Callbacks

A user-written callback should return a nonzero value if the user wishes to stop the optimization and a value of zero otherwise.

For LP, QP, or QCP problems, if the callback returns a nonzero value, the solution process will terminate. If the process was not terminated during the presolve process, the status returned by the function `IloCplex::getStatus` or the routines `CPXsolution` or `CPXgetstat` will be `CPX_STAT_ABORT_USER` (value `13`).

For both LP/QP/QCP and MIP problems, if the LP/QP/QCP callback returns a nonzero value during presolve preprocessing, the optimizer will return the value `CPXERR_PRESLV_ABORT`, and no solution information will be available.

For MIP problems, if the callback returns a nonzero value, the solution process will terminate and the status returned by `IloCplex::getStatus` or `CPXgetstat` will be one of the values in Table 22.2.

Table 22.2 *Status of Nonzero Callbacks for MIPs*

Value	Symbolic constant	Meaning
113	CPXMIP_ABORT_FEAS	current solution integer feasible
114	CPXMIP_ABORT_INFEAS	no integer feasible solution found

Interaction Between Callbacks and ILOG CPLEX Parallel Optimizers

When you use callback routines, and invoke the parallel version of ILOG CPLEX optimizers, you need to be aware that the ILOG CPLEX environment passed to the callback routine corresponds to an individual ILOG CPLEX thread rather than to the original environment created. ILOG CPLEX frees this environment when finished with the thread. This does not affect most uses of the callback function. However, keep in mind that ILOG CPLEX associates problem objects, parameter settings, and message channels with

the environment that specifies them. ILOG CPLEX therefore frees these items when it removes that environment; if the callback uses routines like `CPXcreateprob`, `CPXcloneprob`, or `CPXgetchannels`, those objects remain allocated only as long as the associated environment does. Similarly, setting parameters with routines like `CPXsetintparam` affects settings only within the thread. So, applications that access ILOG CPLEX objects in the callback should use the original environment you created if they need to access these objects outside the scope of the callback function.

Example: Using Callbacks lpex4.c

This example shows you how to use callbacks effectively with routines from the ILOG CPLEX Callable Library. It is based on `lpex1.c`, a program from the ILOG CPLEX *Getting Started* manual. This example about callbacks differs from that simpler one in several ways:

- To make the output more interesting, this example optimizes a slightly different linear program.
- The ILOG CPLEX screen indicator (that is, the parameter `CPX_PARAM_SCRIND`) is not turned on. Only the callback function produces output. Consequently, this program calls `CPXgeterrorstring` to determine any error messages and then prints them. After the `TERMINATE:` label, the program uses separate status variables so that if an error occurred earlier, its error status will not be lost or destroyed by freeing the problem object and closing the ILOG CPLEX environment. Table 22.3 summarizes those status variables.

  Table 22.3 *Status Variables in* `lpex4.c`

Variable	Represents status returned by this routine
`frstatus`	`CPXfreeprob`
`clstatus`	`CPXcloseCPLEX`

- The function `mycallback` at the end of the program is called by the optimizer. This function tests whether the primal simplex optimizer has been called. If so, then a call to `CPXgetcallbackinfo` gets the following information:
  - iteration count;
  - feasibility indicator;
  - sum of infeasibilities (if infeasible);
  - objective value (if feasible).

  The function then prints these values to indicate progress.

- Before the program calls `CPXlpopt`, the default optimizer from the ILOG CPLEX Callable Library, it sets the callback function by calling `CPXsetlpcallbackfunc`. It unsets the callback immediately after optimization.

This callback function offers a model for graphic user interfaces that display optimization progress as well as those GUIs that allow a user to interrupt and stop optimization. If you want to provide your end-user a facility like that to interrupt and stop optimization, then you should make `mycallback` return a nonzero value to indicate the end-user interrupt.

The complete program, `lpex4.c`, appears here and online in the standard distribution.

```
/*---------------------------------------------------------------------------*/
/*  File: examples/src/lpex4.c                                               */
/*  Version 9.0                                                              */
/*---------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2003 by ILOG.                                         */
/*  All Rights Reserved.                                                     */
/*  Permission is expressly granted to use this example in the               */
/*  course of developing applications that use ILOG products.                */
/*---------------------------------------------------------------------------*/

/* lpex4.c - Illustrating the CPLEX callback functionality.

   This is a modification of lpex1.c, where we use a callback
   function to print the iteration info, rather than have CPLEX
   do it. */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

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

#include <string.h>

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

static int
   populatebycolumn  (CPXENVptr env, CPXLPptr lp);

static int CPXPUBLIC
   mycallback (CPXCENVptr env, void *cbdata, int wherefrom,
               void *cbhandle);



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

#define NUMROWS    2
#define NUMCOLS    3
```

```
#define NUMNZ      6

int
main (void)
{
   char     probname[16];  /* Problem name is max 16 characters */

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

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


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

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  The error message will be printed at the end of the
      program. */

   if ( env == NULL ) {
      fprintf (stderr, "Could not open CPLEX environment.\n");
      goto TERMINATE;
   }

   /* Turn *off* output to the screen since we'll be producing it
      via the callback function.  This also means we won't see any
      CPLEX generated errors, but we'll handle that at the end of
      the program. */

   status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_OFF);
   if ( status ) {
      fprintf (stderr,
               "Failure to turn off screen indicator, error %d.\n", status);
      goto TERMINATE;
   }

   /* Create the problem. */
```

```
strcpy (probname, "example");
lp = CPXcreateprob (env, &status, probname);

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, we wouldn't see
   an error message from CPXcreateprob since we turned off the
   CPX_PARAM_SCRIND parameter above.  The only way to see this message
   would be to use the CPLEX message handler, but that clutters up
   the simplicity of this example, which has a point of illustrating
   the CPLEX callback functionality.   */

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

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

status = populatebycolumn (env, lp);

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

status = CPXsetlpcallbackfunc (env, mycallback, NULL);
if ( status ) {
   fprintf (stderr, "Failed to set callback function.\n");
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */

status = CPXsetintparam (env, CPX_PARAM_LPMETHOD, CPX_ALG_PRIMAL);
if ( status ) {
   fprintf (stderr,
            "Failed to set the optimization method, error %d.\n", status);
   goto TERMINATE;
}


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

/* Turn off the callback function.  This isn't strictly necessary,
   but is good practice.  Note that the cast in front of NULL
   is only necessary for some compilers.   */
```

```
   status = CPXsetlpcallbackfunc (env,
             (int (CPXPUBLIC *)(CPXCENVptr, void *, int, void *)) NULL, NULL);

   if ( status ) {
      fprintf (stderr, "Failed to turn off callback function.\n");
      goto TERMINATE;
   }

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


   /* Write the output to the screen. */

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

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

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

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

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

   status = CPXwriteprob (env, lp, "lpex4.lp", NULL);
   if ( status ) {
      fprintf (stderr, "Failed to write LP to disk.\n");
      goto TERMINATE;
   }


TERMINATE:

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

   if ( lp != NULL ) {
      int  frstatus;
```

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

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

   if ( env != NULL ) {
      int  clstatus;
      clstatus = CPXcloseCPLEX (&env);

      if ( clstatus ) {
         fprintf (stderr, "CPXcloseCPLEX failed, error code %d.\n", clstatus);
         if (( !status ) && clstatus )  status = clstatus;
      }
   }

   if ( status ) {
      char  errmsg[1024];

      /* Note that since we have turned off the CPLEX screen indicator,
         we'll need to print the error message ourselves. */

      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, "%s", errmsg);
   }

   return (status);

}  /* END main */


/* This function builds by column the linear program:

      Maximize
       obj: x1 + 2 x2 + 3 x3
      Subject To
       c1: - x1 + x2 + x3 <= 20
       c2: x1 - 3 x2 + x3 <= 30
      Bounds
       0 <= x1 <= 40
      End
 */

static int
populatebycolumn (CPXENVptr env, CPXLPptr lp)
{
   int      status    = 0;
   double   obj[NUMCOLS];
   double   lb[NUMCOLS];
   double   ub[NUMCOLS];
```

```
   char     *colname[NUMCOLS];
   int      matbeg[NUMCOLS];
   int      matind[NUMNZ];
   double   matval[NUMNZ];
   double   rhs[NUMROWS];
   char     sense[NUMROWS];
   char     *rowname[NUMROWS];

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

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

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

   rowname[0] = "c1";
   sense[0]   = 'L';
   rhs[0]     = 20.0;

   rowname[1] = "c2";
   sense[1]   = 'L';
   rhs[1]     = 30.0;

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

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

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

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

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

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

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

   colname[0] = "x1"; colname[1] = "x2";      colname[2] = "x3";

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

TERMINATE:

   return (status);

}  /* END populatebycolumn */
```

```
/* The callback function will print out the Phase of the simplex method,
   the sum of infeasibilities if in Phase 1, or the objective if in Phase 2.
   If any of our requests fails, we'll return an indication to abort.
 */

static int CPXPUBLIC
mycallback (CPXCENVptr env, void *cbdata, int wherefrom, void *cbhandle)
{
   int    status = 0;
   int    phase  = -1;
   double suminf_or_objective;
   int    itcnt = -1;

   if ( wherefrom == CPX_CALLBACK_PRIMAL ) {
      status = CPXgetcallbackinfo (env, cbdata, wherefrom,
                                   CPX_CALLBACK_INFO_ITCOUNT, &itcnt);
      if ( status )  goto TERMINATE;

      status = CPXgetcallbackinfo (env, cbdata, wherefrom,
                                   CPX_CALLBACK_INFO_PRIMAL_FEAS, &phase);
      if ( status )  goto TERMINATE;

      if ( phase == 0 ) {
         status = CPXgetcallbackinfo (env, cbdata, wherefrom,
                                      CPX_CALLBACK_INFO_PRIMAL_INFMEAS,
                                      &suminf_or_objective);
         if ( status )  goto TERMINATE;

         printf ("Iteration %d: Infeasibility measure = %f\n",
                 itcnt, suminf_or_objective);
      }
      else {
         status = CPXgetcallbackinfo (env, cbdata, wherefrom,
                                      CPX_CALLBACK_INFO_PRIMAL_OBJ,
                                      &suminf_or_objective);
         if ( status )  goto TERMINATE;

         printf ("Iteration %d: Objective = %f\n",
                 itcnt, suminf_or_objective);
      }

   }

TERMINATE:

   return (status);

} /* END mycallback */
```

Control Callbacks for IloCplex

Control callbacks allow you to control the branch & cut search during the optimization of MIP problems. The following control callbacks are available for `IloCplex`:

- The node callback allows you to query and optionally overwrite the next node ILOG CPLEX will process during a branch & cut search.
- The solve callback allows you to specify and configure the optimizer option to be used for solving the LP at each individual node.
- The cut callback allows you to add problem-specific cuts at each node.
- The heuristic callback allows you to implement a heuristic that tries to generate a new incumbent from the solution of the LP relaxation at each node.
- The branch callback allows you to query and optionally overwrite the way ILOG CPLEX will branch at each node.
- The incumbent callback allows you to check and optionally reject incumbents found by ILOG CPLEX during the search.

These callbacks are implemented as an extension of the diagnostic callback class hierarchy. This extension is shown below along with the macro names for each of the control callbacks (see *Diagnostic Callbacks* on page 478 for a discussion of how macros and callback implementation classes relate).

```
IloCplex::MIPCallbackI                        ILOMIPCALLBACKn
   |
   +--- IloCplex::NodeCallbackI               ILONODECALLBACKn
   |
   +--- IloCplex::IncumbentCallbackI          ILOINCUMBENTCALLBACKn
   |
   +--- IloCplex::ControlCallbackI            ILOCONTROLCALLBACKn
           |
           +--- IloCplex::BranchCallbackI     ILOBRANCHCALLBACKn
           |
           +--- IloCplex::CutCallbackI        ILOCUTCALLBACKn
           |
          +--- IloCplex::HeuristicCallbackI   ILOHEURISTICCALLBACKn
           |
           +--- IloCplex::SolveCallbackI      ILOSOLVECALLBACKn
```

Again, the callback class hierarchy for Java is exactly the same, but the class names differ, in that there is no `I` at the end. For example, the corresponding Java implementation class for `IloCplex::BranchCallbackI` is `IloCplex.BranchCallback`.

Similar to the class `IloCplex::CallbackI` (`IloCplex.Callback`), the class `IloCplex::ControlCallbackI` (`IloCplex.ControlCallback`) is not provided for deriving user callback classes, but instead for defining the common interface for its derived classes. This interface provides methods for querying information about the current node, such as current bounds or solution information for the current node. See the class `IloCplex::ControlCallbackI (IloCplex.ControlCallback)` in the *ILOG CPLEX Reference Manual* for more information.

Example: Controlling Cuts iloadmipex5.cpp

This example shows how to use the cut callback in the context of solving the *noswot* model. This is a relatively small model from the MIPLIB 3.0 test-set, consisting only of 128 variables. This model is very hard to solve by itself, in fact until the release of ILOG CPLEX version 6.5 it appeared to be unsolvable even after days of computation.

While it is now solvable directly, the computation time is in the order of several hours on state-of-the-art computers. However, cuts can be derived, the addition of which make the problem solvable in a matter of minutes or seconds. These cuts are:

```
x21 - x22 <= 0
x22 - x23 <= 0
x23 - x24 <= 0
2.08*x11 + 2.98*x21 + 3.47*x31 + 2.24*x41 + 2.08*x51 +
0.25*w11 + 0.25*w21 + 0.25*w31 + 0.25*w41 + 0.25*w51 <= 20.25
2.08*x12 + 2.98*x22 + 3.47*x32 + 2.24*x42 + 2.08*x52 +
0.25*w12 + 0.25*w22 + 0.25*w32 + 0.25*w42 + 0.25*w52 <= 20.25
2.08*x13 + 2.98*x23 + 3.47*x33 + 2.24*x43 + 2.08*x53 +
0.25*w13 + 0.25*w23 + 0.25*w33 + 0.25*w43 + 0.25*w53 <= 20.25
2.08*x14 + 2.98*x24 + 3.47*x34 + 2.24*x44 + 2.08*x54 +
0.25*w14 + 0.25*w24 + 0.25*w34 + 0.25*w44 + 0.25*w54 <= 20.25
2.08*x15 + 2.98*x25 + 3.47*x35 + 2.24*x45 + 2.08*x55 +
0.25*w15 + 0.25*w25 + 0.25*w35 + 0.25*w45 + 0.25*w55 <= 16.25
```

These cuts have been derived after interpreting the model as a resource allocation model on five machines with scheduling, horizon constraints and transaction times. The first three cuts break symmetries among the machines, while the others capture minimum bounds on transaction costs. For more information about how these cuts have been found, see *MIP Theory and Practice: Closing the Gap*, available online at http://www.ilog.com/products/optimization/tech/researchpapers.cfm#MIPTheory.

Of course the best way to solve the *noswot* model with these cuts is to simply add the cuts to the model before calling the optimizer. However, for demonstration purposes, this example adds the cuts, using a cut callback, only when they are violated at a node. This cut callback takes a list of cuts as parameter and adds individual cuts whenever they are violated with the current LP solution. Notice, that adding cuts does not change the extracted model, but affects only the internal problem representation of the ILOG CPLEX object.

First consider the C++ version of the callback. In C++, the callback is implemented with the code:

```
ILOCUTCALLBACK3(CtCallback, IloExprArray, lhs, IloNumArray, rhs, IloNum, eps) {
  IloInt n = lhs.getSize();
  for (IloInt i = 0; i < n; ++i) {
    IloNum xrhs = rhs[i];
    if ( xrhs < IloInfinity && getValue(lhs[i]) > xrhs + eps ) {
      IloRange cut;
      try {
        cut = (lhs[i] <= xrhs);
        add(cut).end();
        rhs[i] = IloInfinity;
      }
      catch (...) {
        cut.end();
        throw;
      }
    }
  }
}
```

This defines the class `CtCallbackI` as a derived class of `IloCplex::CutCallbackI` and provides the implementation for its virtual methods `main` and `duplicateCallback`. It also implements a function `CtCallback` that creates an instance of `CtCallbackI` and returns an `IloCplex::Callback` handle for it.

As indicated by the `3` in the macro name, the constructor of `CtCallbackI` takes three parameters, called `lhs`, `rhs`, and `eps`. The constructor stores them as private members to have direct access to them in the callback function, implemented as method `main`. Notice the comma (,) between the type and the argument object in the macro invocation. Here is how the macro expands:

```
class CtCallbackI : public IloCplex::CutCallbackI {
  IloExprArray lhs;
  IloNumArray  rhs;
  IloNum       eps;
public:
  IloCplex::CallbackI* duplicateCallback() const {
    return (new (getEnv()) CtCallbackI(*this));
  }
  CtCallbackI(IloExprArray xlhs, IloNumArray xrhs, IloNum xeps)
    : lhs(xlhs), rhs(xrhs), eps(xeps)
  {}
  void main();
};

IloCplex::Callback CtCallback(IloEnv env,
                              IloExprArray lhs,
                              IloNumArray rhs,
                              IloNum eps) {
  return (IloCplex::Callback(new (env) CtCallbackI(lhs, rhs, eps)));
```

```
}

void CtCallbackI::main() {
  ...
}
```

where the actual implementation code has been substituted with "...". Similar macros are provided for other numbers of parameters ranging from 0 through 7 for all callback classes.

The first parameter `lhs` is an array of expressions, and the parameter `rhs` is an array of values. These parameters are the left-hand side and right-hand side values of cuts of the form `lhs` $\leq$ `rhs` to be tested for violation and potentially added. The third parameter `eps` gives a tolerance by which a cut must at least be violated in order to be added to the problem being solved.

The implementation of this example cut-callback looks for cuts that are violated by the current LP solution of the node where the callback is invoked. It loops over the potential cuts, checking each for violation by querying the value of the `lhs` expression with respect to the current solution. This query calls `getValue` with this expression as a parameter. This value is tested for violation of more than the tolerance parameter `eps` with the corresponding right-hand side value.

> ***Tip:*** *A numerical tolerance is always a wise thing to consider when dealing with any nontrivial model, to avoid certain logical inconsistencies that could otherwise occur due to numerical roundoff. Here the standard ILOG CPLEX simplex feasibility tolerance serves this purpose, to make sure there is consistency with the way ILOG CPLEX is treating the rest of the model.*

If a violation is detected, the callback creates an `IloRange` object to represent the cut: `lhs[i]` $\leq$ `rhs[i]`. It is added to the LP by calling the method `add`. Adding a cut to ILOG CPLEX, unlike extracting a model, only copies the cut into the ILOG CPLEX data structures, without maintaining a notification link between the two. Thus, after a cut has been added, it can be deleted by calling its method `end`. In fact, it should be deleted, as otherwise the memory used for the cut could not be reclaimed. For convenience, method `add` returns the cut that has been added, and thus the application can call `end` directly on the returned `IloRange` object.

It is important that all resources that have been allocated during a callback are freed again before leaving the callback--even in the case of an exception. Here exceptions could be thrown when creating the cut itself or when trying to add it, for example, due to memory exhaustion. Thus, these operations are enclosed in a `try` block to catch all exceptions that may occur. In the case of an exception, the cut is deleted by a call to `cut.end` and whatever exception was caught is then rethrown. Rethrowing the exception can be omitted if you want to continue the optimization without the cut.

After the cut has been added, the application sets the `rhs` value to `IloInfinity` to avoid checking this cut for violation at the next invocation of the callback. Note that it did not

simply remove the i^{th} element of arrays `rhs` and `lhs`, because doing so is not supported if the cut callback is invoked from a parallel optimizer. However, changing array elements is allowed.

Also, for the potential use of the callback in parallel, the variable `xrhs` makes sure that that the same value is used when checking for violation of the cut as when adding the cut. Otherwise, another thread may have set the `rhs` value to `IloInfinity` just between the two actions, and a useless cut would be added. ILOG CPLEX would actually handle this correctly, as it handles adding the same cut from different threads.

The function `makeCuts` generates the arrays `rhs` and `lhs` to be passed to the cut callback. It first declares the array of variables to be used for defining the cuts. Since the environment is not passed to the constructor of that array, an array of 0-variable handles is created. In the following loop, these variable handles are initialized to the correct variables in the `noswot` model which are passed to this function as parameter `vars`. The identification of the variables is done by querying variables names. Once all the variables have been assigned, they are used to create the `lhs` expressions and `rhs` values of the cuts.

The cut callback is created and passed to ILOG CPLEX in the line:

```
cplex.use(CtCallback(env, lhs, rhs, cplex.getParam(IloCplex::EpRHS)));
```

The function `CtCallback` constructs an instance of our callback class `CtCallbackI` and returns an `IloCplex::Callback` handle object for it. This is directly passed to function `cplex.use`.

The Java implementation of the callback is quite similar:

```
public static class Callback extends IloCplex.CutCallback {
   double     eps = 1.0e-6;
   IloRange[] cut;
   Callback(IloRange[] cuts) { cut = cuts; }

   public void main() throws IloException {
      int num = cut.length;
      for (int i = 0; i < num; ++i) {
         if ( cut[i] != null ) {
            double val = getValue(cut[i].getExpr());
            if ( cut[i].getLB() > val+eps || val-eps > cut[i].getUB() ) {
               add(cut[i]);
               cut[i] = null;
            }
         }
      }
   }
}
```

Instead of receiving expressions and right-hand side values, the application directly passes an array of `IloRange` constraints to the callback; the constraints are stored in `cut`. The `main` loops over all cuts and evaluates the constraint expressions at the current solution by calling `getValue(cut[i].getExpr)`. If this value exceeds the constraint bounds by more than `eps`, the cut is added during the search by a call to `add(cut[i])` and `cut[i]` is set to `null` to avoid unneccessarily evaluating it again.

As for the C++ version, the array of cuts passed to the callback is initialized in a separate function `makeCuts`. The callback is then created and used to with the `noswot` cuts by calling.

```
cplex.use(new Callback(makeCuts(cplex, lp)));
```

`IloCplex` provides an easier way to manage such cuts in a case like this, where all cuts can be easily enumerated before starting the optimization. Calling the methods `cplex.addCut` and `cplex.addCuts` allows you to copy the cuts to `IloCplex` before the optimization. Thus, instead of creating and using the callback, a user could have written:

```
cplex.addCuts(makeCuts(var));
```

as shown in example `iloadmipex4.cpp` in the distribution. During branch & cut, ILOG CPLEX will consider adding individual cuts to its representation of the model only if they are violated by a node LP solution in about the same way this example handles them. Whether this or adding the cuts directly to the model gives better performance when solving the model depends on the individual problem.

The complete program, `iloadmipex5.cpp`, appears here and online in the standard distribution.The Java version is found in file `AdMIPex5.java` at the same location.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/iloadmipex5.cpp
// Version 9.0
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2003 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// iloadmipex5.cpp -- Solving noswot by adding cuts in a cut callback
//

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

ILOCUTCALLBACK3(CtCallback, IloExprArray, lhs, IloNumArray, rhs, IloNum, eps) {
   IloInt n = lhs.getSize();
   for (IloInt i = 0; i < n; ++i) {
      IloNum xrhs = rhs[i];
      if ( xrhs < IloInfinity  &&  getValue(lhs[i]) > xrhs + eps ) {
         IloRange cut;
```

```
         try {
            cut = (lhs[i] <= xrhs);
            add(cut).end();
            rhs[i] = IloInfinity;
         }
         catch (...) {
            cut.end();
            throw;
         }
      }
   }
}

void
makeCuts(const IloNumVarArray vars, IloExprArray lhs, IloNumArray rhs) {
   IloNumVar x11, x12, x13, x14, x15;
   IloNumVar x21, x22, x23, x24, x25;
   IloNumVar x31, x32, x33, x34, x35;
   IloNumVar x41, x42, x43, x44, x45;
   IloNumVar x51, x52, x53, x54, x55;
   IloNumVar w11, w12, w13, w14, w15;
   IloNumVar w21, w22, w23, w24, w25;
   IloNumVar w31, w32, w33, w34, w35;
   IloNumVar w41, w42, w43, w44, w45;
   IloNumVar w51, w52, w53, w54, w55;
   IloInt num = vars.getSize();

   for (IloInt i = 0; i < num; ++i) {
      if      ( strcmp(vars[i].getName(), "X11") == 0 ) x11 = vars[i];
      else if ( strcmp(vars[i].getName(), "X12") == 0 ) x12 = vars[i];
      else if ( strcmp(vars[i].getName(), "X13") == 0 ) x13 = vars[i];
      else if ( strcmp(vars[i].getName(), "X14") == 0 ) x14 = vars[i];
      else if ( strcmp(vars[i].getName(), "X15") == 0 ) x15 = vars[i];
      else if ( strcmp(vars[i].getName(), "X21") == 0 ) x21 = vars[i];
      else if ( strcmp(vars[i].getName(), "X22") == 0 ) x22 = vars[i];
      else if ( strcmp(vars[i].getName(), "X23") == 0 ) x23 = vars[i];
      else if ( strcmp(vars[i].getName(), "X24") == 0 ) x24 = vars[i];
      else if ( strcmp(vars[i].getName(), "X25") == 0 ) x25 = vars[i];
      else if ( strcmp(vars[i].getName(), "X31") == 0 ) x31 = vars[i];
      else if ( strcmp(vars[i].getName(), "X32") == 0 ) x32 = vars[i];
      else if ( strcmp(vars[i].getName(), "X33") == 0 ) x33 = vars[i];
      else if ( strcmp(vars[i].getName(), "X34") == 0 ) x34 = vars[i];
      else if ( strcmp(vars[i].getName(), "X35") == 0 ) x35 = vars[i];
      else if ( strcmp(vars[i].getName(), "X41") == 0 ) x41 = vars[i];
      else if ( strcmp(vars[i].getName(), "X42") == 0 ) x42 = vars[i];
      else if ( strcmp(vars[i].getName(), "X43") == 0 ) x43 = vars[i];
      else if ( strcmp(vars[i].getName(), "X44") == 0 ) x44 = vars[i];
      else if ( strcmp(vars[i].getName(), "X45") == 0 ) x45 = vars[i];
      else if ( strcmp(vars[i].getName(), "X51") == 0 ) x51 = vars[i];
      else if ( strcmp(vars[i].getName(), "X52") == 0 ) x52 = vars[i];
      else if ( strcmp(vars[i].getName(), "X53") == 0 ) x53 = vars[i];
      else if ( strcmp(vars[i].getName(), "X54") == 0 ) x54 = vars[i];
```

```
   else if ( strcmp(vars[i].getName(), "X55") == 0 ) x55 = vars[i];
   else if ( strcmp(vars[i].getName(), "W11") == 0 ) w11 = vars[i];
   else if ( strcmp(vars[i].getName(), "W12") == 0 ) w12 = vars[i];
   else if ( strcmp(vars[i].getName(), "W13") == 0 ) w13 = vars[i];
   else if ( strcmp(vars[i].getName(), "W14") == 0 ) w14 = vars[i];
   else if ( strcmp(vars[i].getName(), "W15") == 0 ) w15 = vars[i];
   else if ( strcmp(vars[i].getName(), "W21") == 0 ) w21 = vars[i];
   else if ( strcmp(vars[i].getName(), "W22") == 0 ) w22 = vars[i];
   else if ( strcmp(vars[i].getName(), "W23") == 0 ) w23 = vars[i];
   else if ( strcmp(vars[i].getName(), "W24") == 0 ) w24 = vars[i];
   else if ( strcmp(vars[i].getName(), "W25") == 0 ) w25 = vars[i];
   else if ( strcmp(vars[i].getName(), "W31") == 0 ) w31 = vars[i];
   else if ( strcmp(vars[i].getName(), "W32") == 0 ) w32 = vars[i];
   else if ( strcmp(vars[i].getName(), "W33") == 0 ) w33 = vars[i];
   else if ( strcmp(vars[i].getName(), "W34") == 0 ) w34 = vars[i];
   else if ( strcmp(vars[i].getName(), "W35") == 0 ) w35 = vars[i];
   else if ( strcmp(vars[i].getName(), "W41") == 0 ) w41 = vars[i];
   else if ( strcmp(vars[i].getName(), "W42") == 0 ) w42 = vars[i];
   else if ( strcmp(vars[i].getName(), "W43") == 0 ) w43 = vars[i];
   else if ( strcmp(vars[i].getName(), "W44") == 0 ) w44 = vars[i];
   else if ( strcmp(vars[i].getName(), "W45") == 0 ) w45 = vars[i];
   else if ( strcmp(vars[i].getName(), "W51") == 0 ) w51 = vars[i];
   else if ( strcmp(vars[i].getName(), "W52") == 0 ) w52 = vars[i];
   else if ( strcmp(vars[i].getName(), "W53") == 0 ) w53 = vars[i];
   else if ( strcmp(vars[i].getName(), "W54") == 0 ) w54 = vars[i];
   else if ( strcmp(vars[i].getName(), "W55") == 0 ) w55 = vars[i];
}

lhs.add(x21 - x22);
rhs.add(0.0);

lhs.add(x22 - x23);
rhs.add(0.0);

lhs.add(x23 - x24);
rhs.add(0.0);

lhs.add(2.08*x11 + 2.98*x21 + 3.47*x31 + 2.24*x41 + 2.08*x51 +
        0.25*w11 + 0.25*w21 + 0.25*w31 + 0.25*w41 + 0.25*w51);
rhs.add(20.25);

lhs.add(2.08*x12 + 2.98*x22 + 3.47*x32 + 2.24*x42 + 2.08*x52 +
        0.25*w12 + 0.25*w22 + 0.25*w32 + 0.25*w42 + 0.25*w52);
rhs.add(20.25);

lhs.add(2.08*x13 + 2.98*x23 + 3.47*x33 + 2.24*x43 + 2.08*x53 +
        0.25*w13 + 0.25*w23 + 0.25*w33 + 0.25*w43 + 0.25*w53);
rhs.add(20.25);

lhs.add(2.08*x14 + 2.98*x24 + 3.47*x34 + 2.24*x44 + 2.08*x54 +
        0.25*w14 + 0.25*w24 + 0.25*w34 + 0.25*w44 + 0.25*w54);
rhs.add(20.25);
```

```
   lhs.add(2.08*x15 + 2.98*x25 + 3.47*x35 + 2.24*x45 + 2.08*x55 +
           0.25*w15 + 0.25*w25 + 0.25*w35 + 0.25*w45 + 0.25*w55);
   rhs.add(16.25);
}


int
main(int argc, char** argv)
{
   IloEnv env;
   try {
      IloModel m;
      IloCplex cplex(env);

      IloObjective   obj;
      IloNumVarArray var(env);
      IloRangeArray  con(env);

      env.out() << "reading ../../../examples/data/noswot.mps" << endl;
      cplex.importModel(m, "../../../examples/data/noswot.mps", obj, var, con);

      env.out() << "constructing cut callback ..." << endl;

      IloExprArray lhs(env);
      IloNumArray  rhs(env);
      makeCuts(var, lhs, rhs);
      cplex.use(CtCallback(env, lhs, rhs, cplex.getParam(IloCplex::EpRHS)));

      env.out() << "extracting model ..." << endl;
      cplex.extract(m);

      cplex.setParam(IloCplex::MIPInterval, 1000);
      env.out() << "solving model ...\n";
      cplex.solve();
      env.out() << "solution status is " << cplex.getStatus() << endl;
      env.out() << "solution value  is " << cplex.getObjValue() << endl;
   }
   catch (IloException& ex) {
      cerr << "Error: " << ex << endl;
   }
   env.end();
   return 0;
}
```

23

Goals and Callbacks: a Comparison

Goals and callbacks both provide an API within `IloCplex` to allow you to take control over the branch & cut search for solving MIP models. With one exception, the same functionality is available in both APIs. In fact, the goal API is built on top of callbacks. As a consequence, you cannot use callbacks and goals at the same time. To help you choose which API is more suited to your needs, this section examines commonalities and differences between both.

As pointed out previously, both APIs allow you to control the branch & cut search used by `IloCplex` to solve MIP models. The following points distinguish specific features of this control.

- Checking feasibility
  - With goals, you can discontinue the search at a node by returning a `Fail` goal. Alternatively, you can continue searching, even though an integer feasible solution has been found, by returning another nonempty goal.
  - With callbacks, you can use method `prune` of the branch callback to discontinue the search, and an incumbent callback to accept or reject integer feasible solutions.
- Creating branches
  - With goals, you create branches by using by using `Or` goals with local cut goals as parameters.
  - With callbacks, you create branches by using a branch callback.
- Adding local or global cuts

- With goals, you can add global and local cuts by using global and local cut goals, respectively.
- With callbacks, you need to implement either a cut callback (for global and local cuts) or a branch callback for branching on local cuts

◆ Injecting solution candidates

- With goals, you inject solutions by using a solution goal.
- With callbacks, you need to implement a heuristic callback to inject solutions.

◆ Controlling the node selection strategy

- With goals, you control node selection by applying node evaluators to your search goal.
- With callbacks, you control node selection by using a node callback.

Thus, one of the main differences between goals and callbacks is that with goals, all functionality is available from the `execute` method of the goal, whereas with callbacks, you must implement different callbacks to access different functionality.

As an example, suppose you want to extend a search to satisfy additional constraints that could not conveniently be added as linear constraints to the model.

With callbacks, you need to use an incumbent callback and a branch callback. The incumbent callback has to reject an otherwise integer feasible solution if it violates such an additional constraint. In this case, the branch callback has to follow up with an appropriate branch to enforce the constraint. Since the branch callback function allows branching only on individual variables, the determination of the appropriate branch may be quite difficult for constraints not modelled with linear expressions.

With goals, the feasibility test and the resulting branching can be implemented with a single goal.

The second big difference between goals and callbacks is that with goals you can easily specify different search strategies in different subtrees. To do this, simply provide different search goals as a parameter to the `Or` goal when creating the root nodes for the subtrees in question. To achieve a similar result with callbacks requires an implementation that is too complex for a presentation here.

The only functionality that is not supported via goals is that provided by the `solve` callback. Because of this, the `solve` callbacks can be used at the same time as goals. However, this callback is very rarely used.

In summary, goals can be advantageous if you want to take control over several steps of the branch & cut search simultaneously, or if you want to specify different search strategies in different subtrees. On the other hand, if you only need to control a single aspect of the search—for example, adding cuts—using the appropriate callback may involve a smaller API and thus be quicker and easier to understand and implement.

C H A P T E R

24

User Cut and Lazy Constraint Pools

User cuts are those cuts that a user defines based on information that the user already has about the problem. Lazy constraints are constraints that the user knows are unlikely to be violated, and in consequence, the user wants them applied lazily, that is, only as necessary or not before needed. User cuts can be grouped together in a pool of user cuts. Likewise, lazy constraints can also be grouped into a pool of lazy constriants. This chapter covers those topics.

- *What Are Pools of User Cuts or Lazy Constraints?* on page 507
- *Adding User Cuts and Lazy Constraints* on page 509
- *Deleting User Cuts and Lazy Constraints* on page 511

What Are Pools of User Cuts or Lazy Constraints?

Sometimes, for a MIP formulation, a user may already know a large set of helpful cutting planes (*user cuts*), or can identify a group of constraints that are unlikely to be violated (*lazy constraints*). Simply including these cuts or constraints in the original formulation could make the LP subproblem of a MIP optimization very large or too expensive to solve. Instead, these situations can be handled through the cut callback described in *Advanced MIP Control Interface* on page 513, or by setting up *cut pools* before MIP optimization begins.

The principle in common between these two pools allows the optimization algorithm to perform its computations on a smaller model than it otherwise might, in the hope of delivering faster run times. In either case (whether in the case of pools of user cuts or pools of lazy constraints), the model starts out small, and then potentially grows as members of the pools are added to the model. Both kinds of pool may be used together in solving a MIP model, although that would be an unusual circumstance.

However, there is an important distinction between these two concepts.

Cuts may resemble ordinary constraints, but are conventionally defined to mean those which can change the feasible space of the continuous relaxation but do not rule out any feasible integer solution that the rest of the model permits. A collection of cuts, therefore, involves an element of freedom: whether or not to apply them, individually or collectively, during the optimization of a MIP model; the formulation of the model remains correct whether or not the cuts are included. This degree of freedom means that if valid and necessary constraints are mis-identified by the user and passed to ILOG CPLEX as user cuts, unpredictable and possibly incorrect results could occur.

By contrast, lazy constraints represent simply one portion of the constraint set, and the model would be incomplete (and possibly would deliver incorrect answers) in their absence. ILOG CPLEX always makes sure that lazy constraints are satisfied before producing any solution to a MIP model. Needed lazy constraints are also kept in effect after the MIP optimization terminates, for example, when you change the problem type to fixed-integer and re-optimize with a continuous optimizer.

Another important difference between pools of user cuts and pools of lazy constraints lies in the timing by which these pools are applied. ILOG CPLEX may check user cuts for violation and apply them at any stage of the optimization. Conversely, it does not guarantee to check them at the time an integer-feasible solution candidate has been identified. Lazy constraints are only (and always) checked when an integer-feasible solution candidate has been identified, and of course, any of these constraints that turn out to be violated will then be applied to the full model.

Another way of comparing these two types of pool is to note that the user designates constraints as lazy in the strong hope and expectation that they will not need to be applied, thus saving computation time by their absence from the working problem. In practice, it is relatively costly (for a variety of reasons) to apply a lazy constraint after a violation is identified, and so the user should err on the side of caution when deciding whether a constraint should be marked as lazy. In contrast, user cuts may be more liberally added to a model because ILOG CPLEX is not obligated to use any of them and can apply its own rules to govern their efficient use.

Certain restrictions apply to these pools unless you are using Concert Technology. (Concert Technology will automatically handle these ILOG CPLEX parameter settings for you.) If either of these conditions is violated, the error `CPXERR_PRESOLVE_BAD_PARAM` will be issued when the MIP optimizer is called.

- When a user cut pool is present, the parameter `CPX_PARAM_ PRELINEAR` (`PreLinear` in Concert Technology) must be set to zero.
- When a lazy constraint pool is present, the parameter `CPX_PARAM_ REDUCE` (`Reduce` in Concert Technology) must be set to either 0 (zero) or 1 (one), in order that dual reductions **not** be performed by presolve during preprocessing.

Adding User Cuts and Lazy Constraints

You may add user cuts or lazy constraints through `add` routines in the Component Libraries or via LP and SAV files.

Component Libraries

The following routines will add to the user cut pool.

- The CPLEX Callable Library routine is `CPXaddusercuts`.
- The Concert Technology routine is `IloCplex::addUserCuts`.

The following routines will add to the lazy constraint pool.

- The CPLEX Callable Library routine is `CPXaddlazyconstraints`.
- The Concert Technology routine is `IloCplex::addLazyConstraints`.

Reading LP and SAV Files

User cuts and lazy constraints may also be specified in LP-format and SAV-format files, and so may be read:

- With the Interactive Optimizer.
- Through the routines `CPXreadcopyprob` and `IloCplex::importModel`.

General Syntax

The general syntax rules for LP format given in the reference manual *ILOG CPLEX File Formats* apply to user cuts and lazy constraints.

- The user cuts section or sections must be preceded by the keywords `USER CUTS`.
- The lazy constraints section or sections must be preceded by the keywords `LAZY CONSTRAINTS`.

These sections, and the ordinary constraints section preceded by the keywords `SUBJECT TO`, can appear in any order and can be present multiple times, as long as they are

placed after the objective function section and before any of the keywords BOUNDS, GENERALS, BINARIES, SEMI-CONTINUOUS or END.

Example

Here is an example of a file containing ordinary constraints and lazy constraints.

```
Maximize
 obj: 12 x1 + 5 x2 + 15 x3 + 10 x4
Subject To
 c1: 5 x1 + x2 + 9 x3 + 12 x4 <= 15
Lazy Constraints
 l1: 2 x1 + 3 x2 + 4 x3 + x4 <= 10
 l2: 3 x1 + 2 x2 + 4 x3 + 10 x4 <= 8
Bounds
 0 <= x1 <= 5
 0 <= x2 <= 5
 0 <= x3 <= 5
 0 <= x4 <= 5
Generals
 x1  x2  x3  x4
End
```

ILOG CPLEX stores user cuts and lazy constraints in memory separately from ordinary constraints, and any optional constraint names provided for them in the input (that is, preceded by a colon :) are not retained. In other words, the optional constraint names in LP format for user cuts and lazy constraints are discarded.

Writing LP and SAV Files

When writing LP or SAV format files, user cuts and lazy constraints added through their respective add routines or read from LP or SAV format files will be included in the output files. In LP-format files:

- User cuts will be given names of the form ux where x is an index number starting at 1.
- Lazy constraints will be given names of lx where x is again an index number starting at 1.

Using the Interactive Optimizer

User cuts and lazy constraints will appear when the command `display problem all` is issued in the Interactive Optimizer. User cuts and lazy constraints can also be added to an existing problem with the add command of the Interactive Optimizer.

Deleting User Cuts and Lazy Constraints

The user cut and lazy constraint pools are cleared by calling the routines `CPXfreeusercuts` and `CPXfreelazyconstraints`. Clearing the pools will not change the MIP solution.

The Concert Technology routines are `IloCplex::clearUserCuts` and `IloCplex::clearLazyConstraints`.

Clearing a pool means that the user cuts and lazy constraints in the pool will be removed and will not be applied the next time MIP optimization is called, and that the solution to the MIP (if one exists) is still available. Although any existing solution is still feasible, it may no longer be optimal because of this change in the constraints.

C H A P T E R

25

Advanced MIP Control Interface

In this manual, *Using Callbacks* on page 477 introduces callbacks, their purpose, and conventions. This chapter documents the CPLEX advanced MIP control interface, describing callbacks in greater detail. It assumes that you are already familiar with that introduction to callbacks in general, and it includes sections about:

- *Introduction to MIP Callbacks* on page 514
- *Heuristic Callback* on page 515
- *Cut Callback* on page 516
- *Branch Selection Callback* on page 517
- *Incumbent Callback* on page 518
- *Node Selection Callback* on page 519
- *Solve Callback* on page 519

These callbacks allow sophisticated users to control the details of the branch & cut process. Specifically, users can choose the next node to explore, choose the branching variable, add their own cutting planes, place additional restrictions on integer solutions, or insert their own heuristic solutions. These functions are meant for situations where other tactics to improve CPLEX performance on a hard MIP problem, such as non-default parameter settings or priority orders, have failed. See *Troubleshooting MIP Performance Problems* on page 335 for more information about MIP parameters and priority orders.

Users of the advanced MIP control interface can work with the variables of the presolved problem or, by following a few simple rules, can instead work with the variables of the original problem.

> ***Tip:*** The advanced MIP control interface relies heavily on the advanced presolve capabilities. We suggest that the reader become familiar with Chapter 26, *Advanced Presolve Routines*, before reading this chapter.

Control callbacks in the ILOG Concert Technology CPLEX Library use original model variables. These callbacks are fully documented in the *ILOG CPLEX Reference Manual*.

Introduction to MIP Callbacks

As the reader is no doubt familiar, the process of solving a mixed integer programming problem involves exploring a tree of linear programming relaxations. CPLEX repeatedly selects a node from the tree, solves the LP relaxation at that node, attempts to generate cutting planes to cut off the current solution, invokes a heuristic to try to find an integer feasible solution "close" to the current relaxation solution, selects a branching variable (an integer variable whose value in the current relaxation is fractional), and finally places the two nodes that result from branching up or down on the branching variable back into the tree.

The CPLEX Mixed Integer Optimizer includes methods for each of the important steps listed above (node selection, cutting planes, heuristic, branch variable selection, incumbent replacement). While default CPLEX methods are generally effective, and parameters are available to choose alternatives if the defaults are not working for a particular problem, there are rare cases where a user may wish to influence or even override CPLEX methods. CPLEX provides a callback mechanism to allow the user to do this. If the user installs a callback routine, CPLEX calls this routine during the branch & cut process to allow the user to intervene. CPLEX callback functions are thread-safe for use in parallel (multiple CPU) applications.

Before describing the callback routines, we first discuss an important issue related to presolve that the user should be aware of. Most of the decisions made within MIP relate to the variables of the problem. The heuristic, for example, finds values for all the variables in the problem that produce a feasible solution. Similarly, branching chooses a variable on which to branch. When considering user callbacks, the difficulty that arises is that the user is familiar with the variables in the original problem, while the branch & cut process is performed on the presolved problem. Many of the variables in the original problem may have been modified or removed by presolve.

CPLEX provides two options for handling the problem of mapping from the original problem to the presolved problem. First, the user may work directly with the presolved problem and presolved solution vectors. This is the default. While this option may at first

appear unwieldy, note that the Advanced Presolve Interface allows the user to map between original variables and presolved variables. The downside to this option is that the user has to manually invoke these advanced presolve routines. The second option is to set CPX_PARAM_MIPCBREDLP to CPX_OFF (0), thus requesting that the callback routines work exclusively with original variables. CPLEX automatically translates the data between original and presolved data. While the second option is simpler, the first provides more control. These two options will be revisited at several points in this chapter.

Heuristic Callback

The first user callback we consider is the heuristic callback. The first step in using this callback is to call CPXsetheuristiccallbackfunc, with a pointer to a callback function and optionally a pointer to user private data as arguments. We refer the reader to advanced example admipex2.c for further details of how this callback is used. Once this routine has been called, CPLEX calls the user callback function at every viable node in the branch & cut tree (we call a node viable if its LP relaxation is feasible and its relaxation objective value is better than that of the best available integer solution). The user callback routine is called with the solution vector for the current relaxation as input. The callback function should return a feasible solution vector, if one is found, as output.

The advanced MIP control interface provides several routines that allow the user callback to gather information that may be useful in finding heuristic solutions. The routines CPXgetcallbackgloballb and CPXgetcallbackglobalub, for example, return the tightest known global lower and upper bounds on all the variables in the problem. No feasible solution whose objective is better than that of the best known solution can violate these bounds. Similarly, the routines CPXgetcallbacknodelb and CPXgetcallbacknodeub return variable bounds at this node. These reflect the bound adjustments made during branching. The routine CPXgetcallbackincumbent returns the current incumbent - the best known feasible solution. The routine CPXgetcallbacklp returns a pointer to the MIP problem (presolved or unpresolved, depending on the CPX_PARAM_MIPCBREDLP parameter). This pointer can be used to obtain various information about the problem (variable types, etc.), or as an argument for the advanced presolve interface if the user wishes to manually translate between presolved and unpresolved values. In addition, the callback can use the cbdata parameter passed to it, along with routine CPXgetcallbacknodelp, to obtain a pointer to the node relaxation LP. This can be used to access desired information about the relaxation (row count, column count, etc.). Note that in both cases, the user should never use the pointers obtained from these callbacks to modify the associated problems.

As noted earlier, the CPX_PARAM_MIPCBREDLP parameter influences the arguments to the user callback routine. If this parameter is set to its default value of CPX_ON (1), the solution vector returned to the callback, and any feasible solutions returned by the callback, are presolved vectors. They contain one value for each variable in the presolved problem. The

same is true of the various callback support routines (CPXgetcallbackgloballb, etc.). If the parameter is set to CPX_OFF (0), all these vectors relate to variables of the original problem. Note that this parameter should not be changed in the middle of an optimization.

The user should be aware that the branch & cut process works with the presolved problem, so the code will incur some cost when translating from presolved to original values. This cost is usually small, but can sometimes be significant.

We should also note that if a user wishes to solve linear programs as part of a heuristic callback, the user must make a copy of the node LP (for example, using CPXcloneprob). The user should not modify the CPLEX node LP.

Cut Callback

The next example we consider is the user cut callback routine. The user calls CPXsetcutcallbackfunc to set a cut callback, and the user's callback routine is called at every viable node of the branch & cut tree. We refer the reader to admipex5.c for a detailed example.

A likely sequence of events once the user callback function is called is as follows. First, the routine calls CPXgetcallbacknodex to get the relaxation solution for the current node. It possibly also gathers other information about the problem (through CPXgetcallbacklp, CPXgetcallbackgloballb, etc.) It then calls a user separation routine to identify violated user cuts. These cuts are then added to the problem by calling CPXcutcallbackadd, and the callback returns. Local cuts, that is, cuts that apply to the subtree of which the current node is the root, can be added by the routine CPXcutcallbackaddlocal.

At this point, it is important to draw a distinction between the two different types of constraints that can be added through the cut callback interface. The first type is the traditional MIP cutting plane, which is a constraint that can be derived from other constraints in the problem and does not cut off any integer feasible solutions. The second is a "lazy constraint", which is a constraint that can not be derived from other constraints and potentially cuts off integer feasible solutions. Either type of constraint can be added through the cut callback interface.

As with the heuristic callback, the user can choose whether to work with presolved values or original values. If the user chooses to work with original values, a few parameters must be modified. If the user adds only cutting planes to the original problem, the user must set advanced presolve parameter CPX_PARAM_PRELINEAR to CPX_OFF (0). This parameter forbids certain presolve reductions that make translation from original values to presolved values impossible.

If the user adds any lazy constraints, the user must turn off dual presolve reductions (using the CPX_PARAM_REDUCE parameter). The user must think carefully about whether constraints added through the cut interface are implied by existing constraints, in which case

dual presolve reductions may be left on, or whether they are not, in which case dual reductions are forbidden.

ILOG Concert Technology users should use the class `IloCplex::LazyConstraintCallbackI` when adding lazy constraints, and the class `IloCplex::UserCutCallbackI` when adding cutting planes. Dual reductions and/or non-linear reductions then will be turned off automatically.

One scenario that merits special attention is when the user knows a large set of cuts a priori. Rather than adding them to the original problem, the user may instead wish to add them only when violated. The CPLEX advanced MIP control interface provides more than one mechanism for accomplishing this. The first and probably most obvious at this point is to install a user callback that checks each cut from the user set at each node, adding those that are violated. The user can do this either by setting `CPX_PARAM_MIPCBREDLP` to `CPX_OFF` to work with the original problem in the cut callback, or by using the Advanced Presolve Interface to translate the cuts on the original problem to cuts on the presolved problem, and then use the presolved cuts in the cut callback.

Another, perhaps simpler alternative is to add the cuts or constraints to cut pools before optimization begins. Pools are discussed in *User Cut and Lazy Constraint Pools* on page 507.

Branch Selection Callback

The next callback to consider is the branch variable selection callback.

After `CPXsetbranchcallbackfunc` is called with a pointer to a user callback routine, the user routine is called whenever CPLEX makes a branching decision. CPLEX indicates which variable has been chosen for branching and allows the user to modify that decision. The user may specify the number of children for the current node (between 0 and 2), and the set of bounds or constraints that are modified for each child through one of the routines `CPXbranchcallbackbranchbds`, `CPXbranchcallbackbranchconstraints`, or `CPXbranchcallbackbranchgeneral`. The children are explored in the order that they are returned. The branch callback routine is called for all viable nodes. In particular, it will be called for nodes that have zero integer infeasibilities; in this case, CPLEX will not have chosen a branch variable, and the default action will be to discard the node. The user can choose to branch from this node and in this way impose additional restrictions on integer solutions.

For example, a user branch routine may call `CPXgetcallbacknodeintfeas` to identify branching candidates, call `CPXgetcallbackpseudocosts` to obtain pseudo-cost information on these variables, call `CPXgetcallbackorder` to get priority order information, make a decision based on this and perhaps other information, and then respond

that the current node will have two children, where one has a new lower bound on the branch variable and the other has a new upper bound on that variable.

Alternatively, the branch callback routine can be used to sculpt the search tree by pruning nodes or adjusting variable bounds. Choosing zero children for a node prunes that node, while choosing one node with a set of new variable bounds adjusts bounds on those variables for the entire subtree rooted at this node. Note that the user must be careful when using this routine for anything other than choosing a different variable to branch on. Pruning a valid node or placing an invalid bound on a variable can prune the optimal solution.

We should point out one important detail associated with the use of the `CPX_PARAM_MIPCBREDLP` parameter in a branch callback. If this parameter is set to `CPX_OFF` (0), the user can choose branch variables (and add bounds) for the original problem. However, not every fractional variable is actually available for branching. Recall that some variables are replaced by linear combinations of other variables in the presolved problem. Since branching involves adding new bounds to specific variables in the presolved problem, a variable must be present in the presolved problem for it to be branched on. The user should use the `CPXgetcallbacknodeintfeas` routine from the Advanced Presolve Interface to find branching candidates (those for which `CPXgetcallbacknodeintfeas` returns `CPX_INTEGER_INFEASIBLE`). The `CPXcopyprotected` routine can be used to prevent presolve from removing specific variables from the presolved problem. (In Concert Technology, this issue is handled for you automatically.) While restricting branching may appear to limit your ability to solve a problem, in fact a problem can always be solved to optimality by branching only on the variables of the presolved problem.

Incumbent Callback

The incumbent callback is used to reject integer feasible solutions that do not meet additional restrictions the user may wish to impose. The user callback routine will be called each time a new incumbent solution has been found, including when solutions are provided by the user's heuristic callback routine. The user callback routine is called with the new solution as input. The callback function should return a value that indicates whether or not the new solution should replace the incumbent solution.

As with other MIP control callback routines, the `CPX_PARAM_MIPCBREDLP` parameter influences the arguments to the user callback routine. If this parameter is set to its default value of `CPX_ON (1)`, the solution vector that is input to the callback is a presolved vector. It contains one value for each variable in the presolved problem. The same is true of the various callback support routines (`CPXcallbackglobalub`, and so forth.). If the parameter is set to `CPX_OFF (0)`, all these vectors relate to the variables of the original problem. Note that this parameter should not be changed in the middle of an optimization.

Node Selection Callback

The user can influence the order in which nodes are explored by installing a node selection callback (through `CPXsetnodecallbackfunc`). When CPLEX chooses the node to explore next, it will call the user callback routine, with CPLEX's choice as an argument. The callback has the option of modifying this choice.

Solve Callback

The final callback we consider is the solve callback. By calling `CPXsetsolvecallbackfunc`, the user instructs CPLEX to call a user function rather than the CPLEX choice (dual simplex by default) to solve the linear programming relaxations at each node of the tree. Advanced example `admipex6.c` gives an example of how this callback might be used.

> ***Note:*** *We expect the most common use of this callback will be to craft a customer solution strategy out of the set of available CPLEX algorithms. For example, a user might create a hybrid strategy that checks for network status, calling* `CPXhybnetopt` *instead of* `CPXdualopt` *when it finds it.*

26

Advanced Presolve Routines

This chapter explains how to use the advanced presolve routines. The topics are:

- *Introduction to Presolve* on page 521
- *Restricting Presolve Reductions* on page 523
- *Manual Control of Presolve* on page 526
- *Modifying a Problem* on page 528

Introduction to Presolve

We begin our discussion of the advanced presolve interface with a quick introduction to presolve. Most of the information in this section will be familiar to people who are interested in the advanced interface, but we encourage everyone to read through it nonetheless.

As most CPLEX users know, presolve is a process whereby the problem input by the user is examined for logical reduction opportunities. The goal is to reduce the size of the problem passed to the requested optimizer. A reduction in problem size typically translates to a reduction in total run time (even including the time spent in presolve itself).

Consider `scorpion.mps` from NETLIB as an example:

```
CPLEX> disp pr st
Problem name: scorpion.mps
Constraints        :     388  [Less: 48,  Greater: 60,  Equal: 280]
Variables          :     358
Constraint nonzeros:    1426
Objective  nonzeros:     282
RHS        nonzeros:      76
CPLEX> optimize
Tried aggregator 1 time.
LP Presolve eliminated 138 rows and 82 columns.
Aggregator did 193 substitutions.
Reduced LP has 57 rows, 83 columns, and 327 nonzeros.
Presolve time =    0.00 sec.

Iteration log . . .
Iteration:     1   Dual objective     =           317.965093

Dual - Optimal:  Objective =    1.8781248227e+03
Solution time =    0.01 sec.  Iterations = 54 (0)
```

CPLEX is presented with a problem with 388 constraints and 358 variables, and after presolve the dual simplex method is presented with a problem with 57 constraints and 83 variables. Dual simplex solves this problem and passes the solution back to the presolve routine, which then unpresolves the solution to produce a solution to the original problem. During this process, presolve builds an entirely new ‘presolved’ problem and stores enough information to translate a solution to this problem back to a solution to the original problem. This information is hidden within the user's problem (in the CPLEX LP problem object, for Callable Library users) and was inaccessible to the user in CPLEX releases prior to 7.0.

The presolve process for a mixed integer program is similar, but has a few important differences. First, the actual presolve reductions differ. Integrality restrictions allow CPLEX to perform several classes of reductions that are invalid for continuous variables. A second difference is that the MIP solution process involves a series of linear program solutions. In the MIP branch & cut tree, a linear program is solved at each node. MIP presolve is performed once, at the beginning of the optimization (unless the `CPX_PARAM_RELAXPREIND` parameter is set, in which case the root relaxation is presolved a second time), and all of the node relaxation solutions use the presolved problem. Again, presolve stores the presolved problem and the information required to convert a presolved solution to a solution for the original problem within the LP problem object. Again, this information was inaccessible to the user in CPLEX releases prior to version 7.0.

A Proposed Example

Now consider an application where the user wishes to solve a linear program using the simplex method, then modify the problem slightly and solve the modified problem. As an example, let's say a user wishes to add a few new constraints to a problem based on the results of the first solution. The second solution should ideally start from the basis of the first, since starting from an advanced basis is usually significantly faster if the problem is only modified slightly.

Unfortunately, this scenario presents several difficulties. First, presolve must translate the new constraints on the original problem into constraints on the presolved problem. Presolve in releases prior to 7.0 could not do this. In addition, the new constraints may invalidate earlier presolve reductions, thus rendering the presolved problem useless for the reoptimization. (There is an example in *Restricting Presolve Reductions* on page 523.) Presolve in releases prior to 7.0 had no way of disabling such reductions. In the prior releases, a user could either restart the optimization on the original, unpresolved problem or perform a new presolve on the modified problem. In the former case, the reoptimization does not benefit from the reduction of the problem size by presolve. In the latter, the second optimization does not have the benefit of an advanced starting solution.

The advanced presolve interface can potentially make this and many other sequences of operations more efficient. It provides facilities to restrict the set of presolve reductions performed so that subsequent problem modifications can be accommodated. It also provides routines to translate constraints on the original problem to constraints on the presolved problem, so new constraints can be added to the presolved problem. In short, it provides a variety of capabilities.

When considering mixed integer programs, the advanced presolve interface plays a very different role. The branch & cut process needs to be restarted from scratch when the problem is even slightly modified, so preserving advanced start information isn't useful. The main benefit of the advanced presolve interface for MIPs is that it allows a user to translate decisions made during the branch & cut process (and based on the presolved problem) back to the corresponding constraints and variables in the original problem. This makes it easier for a user to control the branch & cut process. Details on the advanced MIP callback interface are provided in *Advanced MIP Control Interface* on page 31.

Restricting Presolve Reductions

As mentioned in *Introduction to Presolve* on page 521, some presolve reductions are invalidated when a problem is modified. The advanced presolve interface therefore allows a user to tell presolve what sort of modifications will be made in the future, so presolve can avoid possibly invalid reductions. These considerations only apply to linear programs. Any modifications of QP or QCP models will cause ILOG CPLEX to discard the presolved model.

Example: Adding Constraints to the First Solution

Consider adding a constraint to a problem after solving it. Imagine that you want to optimize a linear program:

Primal:		**Dual:**	
max	$-x1 + x2 + x3$	**min**	$6y1 + 5y2$

st	*x1*	+	*x2*	+	*2x3*	≤	*6*	**st**	*y1*					≥	*-1*
			x2	+	*x3*	≤	*5*		*y1*	+	*y2*			≥	*1*
							0		*2y1*	+	*y2*			≥	*1*
	x1,		*x2,*		*x3*	≥	*0*		*y1,*		*y2,*		*y3*	≥	*0*

Note that the first constraint in the dual ($y1 \geq -1$) is redundant. Presolve can use this information about the dual problem (and complementary slackness) to conclude that variable *x1* can be fixed to 0 and removed from the presolved problem. While it may be intuitively obvious from inspection of the primal problem that *x1* can be fixed to 0, it is important to note that dual information (redundancy of the first dual constraint) is used to formally prove it.

Now consider the addition of a new constraint $x2 \leq 5x1$:

Primal:								**Dual:**							
max	*-x1*	+	*x2*	+	*x3*			**min**	*6y1*	+	*5y2*				
st	*x1*	+	*x2*	+	*2x3*	≤	*6*	**st**	*y1*			–	*5y3*	≥	*-1*
			x2	+	*x3*	≤	*5*		*y1*	+	*y2*	+	*y3*	≥	*1*
	- 5x1	+	*x2*			≤	*0*		*2y1*	+	*y2*			≥	*1*
	x1,		*x2,*		*x3*	≥	*0*		*y1,*		*y2,*		*y3*	≥	*0*

Our goal is to add the appropriate constraint to the presolved problem and reoptimize. Note, however, that the dual information presolve used to fix *x1* to 0 was changed by the addition of the new constraint. The first constraint in the dual is no longer guaranteed to be redundant, so the original fixing is no longer valid (the optimal solution is now $x1=1, x2=5, x3=0$). As a result, CPLEX is unable to use the presolved problem to reoptimize.

We classify presolve reductions into several classes: those that rely on primal information, those that rely on dual information, and those that rely on both. Future addition of new constraints, modifications to objective coefficients, and tightening of variable bounds (a special class of adding new constraints) require the user to turn off dual reductions. Introduction of new columns, modifications to right-hand-side values, and relaxation of variable bounds (a special case of modifying right-hand-side values) require the user to turn off primal reductions.

These reductions are controlled through the `CPX_PARAM_REDUCE` parameter. The parameter has four possible settings. The default value `CPX_PREREDUCE_PRIMALANDDUAL` (3) indicates that presolve can rely on primal and dual information. With setting `CPX_PREREDUCE_DUALONLY` (2), presolve only uses dual information, with setting `CPX_PREREDUCE_PRIMALONLY` (1) it only uses primal information, and with setting `CPX_PREREDUCE_NO_PRIMALORDUAL` (0) it uses neither (which is equivalent to turning presolve off).

Setting the CPX_PARAM_REDUCE parameter has one additional effect on the optimization. Normally, the presolved problem and the presolved solution are freed at the end of an optimization call. However, when CPX_PARAM_REDUCE is set to a value other than its default, CPLEX assumes that the problem will subsequently be modified and reoptimized. It therefore retains the presolved problem and any presolved solution information (internally to the LP problem object). If the user has set CPX_PARAM_REDUCE and is finished with problem modification, the user can call CPXfreepresolve to free the presolved problem and reclaim the associated memory. The presolved problem is automatically freed when the user calls CPXfreeprob on the original problem.

We should note that cutting planes in mixed integer programming are handled somewhat differently than one might expect. If a user wishes to add his own cuts during the branch & cut process (through CPXaddusercuts or CPXcutcallbackadd), it may appear necessary to turn off dual reductions to accommodate them. However, for reasons that are complex and beyond the scope of this discussion, dual reductions can be left on. The reasons relate to the fact that valid cuts never exclude integer feasible solutions, so dual reductions performed for the original problem are still valid after cutting planes are applied. However, a small set of reductions does need to be turned off. Recall that presolve must translate a new constraint on the original problem into a constraint on variables in the presolved problem. Most reductions performed by CPLEX presolve replace variables with linear expressions of zero or more other variables (plus a constant). A few do not. These latter reductions make it impossible to perform the translation to the presolved problem. Set CPX_PARAM_PRELINEAR to 0 to forbid these latter reductions.

Restricting the type of presolve reductions will also allow presolve to conclude more about infeasible and/or unbounded problems. Under the default setting of CPX_PARAM_REDUCE, presolve can only conclude that a problem is infeasible and/or unbounded. If CPX_PARAM_REDUCE is set to CPX_PREREDUCE_PRIMALONLY (1), presolve can conclude that a problem is primal infeasible with return status CPXERR_PRESLV_INF. If CPX_PARAM_REDUCE is set to CPX_PREREDUCE_DUALONLY (2), presolve can conclude that a problem is primal unbounded (if it is primal feasible) with return status CPXERR_PRESLV_UNBD.

Reminder: *The previous paragraph applies to CPXpresolve,* ***not*** *CPXlpopt.*

A final facility that modifies the set of reductions performed by presolve is the CPXcopyprotected routine. The user provides as input a list of variables in the original problem that should survive presolve (that is, should exist in the presolved problem). Presolve will avoid reductions that remove those variables, with one exception. If a protected variable can be fixed, presolve will still remove it from the problem. This command is useful in cases where the user wants to explicitly control some aspect of the branch & cut process (for example, through the branch callback) using knowledge about those variables.

Manual Control of Presolve

While presolve was a highly automated and transparent procedure in releases of CPLEX prior to 7.0, releases 7.0 and above give the user significant control over when presolve is performed and what is done with the result. This section discusses these added control facilities. Recall that the functions mentioned here are documented in detail, including arguments and return values, in the reference manual.

The first control function provided by the advanced presolve interface is `CPXpresolve`, which manually invokes presolve on the supplied problem. Once this routine is called on a problem, the problem has a presolved problem associated with it. Subsequent calls to optimization routines (`CPXprimopt`, `CPXdualopt`, `CPXbaropt`, `CPXmipopt`) will use this presolved problem without repeating the presolve, provided no operation that discards the presolved problem is performed in the interim. The presolved problem is automatically discarded if a problem modification is performed that is incompatible with the setting of `CPX_PARAM_REDUCE` (further information is provided in *Modifying a Problem* on page 38).

By using the `CPX_PARAM_REDUCE` to restrict the types of presolve reductions, CPLEX can make use of the optimal basis to the presolved problem. If you set `CPX_PARAM_REDUCE` to restrict presolve reductions, then make problem modifications that don't invalidate those reductions, CPLEX will automatically use the optimal basis to the presolved problem. On the other hand, if the nature of the problem modifications is such that you cannot set `CPX_PARAM_REDUCE`, you can still perform an advanced start by making the modifications, calling `CPXpresolve` to create the new presolved problem, then calling `CPXcopystart`, passing the original model and any combination of primal and dual solutions. With nondefault settings of `CPX_PARAM_REDUCE`, CPLEX will crush the solutions and use them to construct a starting basis for the presolved model. If you are continuing with primal simplex, only providing a primal starting vector will usually perform better.

We should point out a few of the subtleties associated with using `CPXcopystart` to start an optimization from an advanced, presolved solution. This routine only creates a presolved solution vector if the presolved problem is already present (either because the user called `CPXpresolve` or because the user turned off some presolve reductions through `CPX_PARAM_REDUCE` and then solved a problem). The earlier sequence would not have started from an advanced solution if `CPXcopystart` had been called before `CPXpresolve`. Another important detail about `CPXcopystart` is that it crushes primal and/or dual solutions, not bases. It then uses the crushed solutions to choose a starting basis. If you have a basis, you need to compute optimal primal and dual solutions (using `CPXcopybase` and then `CPXprimopt`), extract them, and then call `CPXcopystart` with them to use the corresponding advanced solution. In general, starting with both a primal and dual solution is preferable to starting with one or the other. One other thing to note about `CPXcopystart` is that the primal and dual slack (slack and dj) arguments are optional. The routine will compute slack values if none are provided.

Recall that you can set the parameter CPX_PARAM_ADVIND to 2 in order to use advanced starting information together with presolve. At this setting, CPLEX will use starting information provided to it with CPXcopystart or CPXcopybase when it solves an LP with the primal or dual simplex optimizer in the following way. If no presolved model is available, presolve is invoked. Then the starting information is crushed and installed in the presolved model. Finally, the presolved model is solved from the advanced (crushed) starting point.

Another situation where a user might want to use CPXpresolve is if the user wishes to gather information about the presolve, possibly in preparation for using the advanced MIP callback routines to control the branch & cut process. Once CPXpresolve has been called, the user can then call CPXgetprestat to obtain information about the reductions performed on the problem. This function provides information, for each variable in the original problem, on whether the variable was fixed and removed, aggregated out, removed for some other reason, or is still present in the reduced problem. It also gives information, for each row in the original problem, on whether it was removed due to redundancy, aggregated out, or is still present in the reduced problem. For those rows and columns that are present in the reduced problem, this function provides a mapping from original row/column number to row/column number in the reduced problem, and vice-versa.

Another situation where a user might want to use CPXpresolve is to work directly on the presolved problem. By calling CPXgetredlp immediately after CPXpresolve, the user can obtain a pointer to the presolved problem. For an example of how this might be used, the user could call routines CPXcrushx and CPXcrushpi to presolve primal and dual solution vectors, call CPXgetredlp to get access to the presolved problem, then use CPXcopystart to copy the presolved solutions into the presolved problem, then optimize the problem, and finally call routines CPXuncrushx and CPXuncrushpi—CPXqpuncruspi for QPs—to unpresolve solutions from the presolved problem, creating solutions for the original problem.

Please note that CPXgetredlp provides the user access to internal CPLEX data structures. The presolved problem **must not** be modified by the user. If the user wishes to manipulate the reduced problem, the user should make a copy of it (using CPXcloneprob) and manipulate the copy instead.

The advanced presolve interface provides another call that is useful when working directly with the presolved problem (either through CPXgetredlp or through the advanced MIP callbacks). The CPXcrushform call translates a linear expression in the original problem into a linear expression in the presolved problem. The most likely use of this routine is to add user cuts to the presolved problem during a mixed integer optimization. The advanced presolve interface also provides the reverse operation. The CPXuncrushform routine translates a linear expression in the presolved problem into a linear expression in the original problem.

A limited presolve analysis is performed by CPXbasicpresolve and by the Concert Technology method IloCplex::basicPresolve. This function determines which rows

are redundant and computes strengthened bounds on the variables. This information can be used to derive some types of cuts that will tighten the formulation, to aid in formulation by pointing out redundancies, and to provide upper bounds for variables that must have them, like semi-continuous variables. `CPXbasicpresolve` does not create a presolved problem.

The interface allows the user to manually free the memory associated with the presolved problem using routine `CPXfreepresolve`. The next optimization call (or call to `CPXpresolve`) recreates the presolved problem.

Modifying a Problem

This section briefly discusses the mechanics of modifying a problem after presolve has been performed. This discussion applies only to linear programs; it does **not** apply to quadratic programs, quadratically constrained programs, nor mixed integer programs.

As noted earlier, the user must indicate through the `CPX_PARAM_REDUCE` parameter the types of modifications that are going to be performed to the problem. Recall that if primal reductions are turned off, the user can add variables, change the right-hand-side vector, or loosen variable bounds without losing the presolved problem. These changes are made through the standard problem modification interface (`CPXaddcols`, `CPXchgrhs`, and `CPXchgbds`).

Recall that if dual reductions are turned off, the user can add constraints to the problem, change the objective function, or tighten variable bounds. Variable bounds are tightened through the standard interface (`CPXchgbds`). The addition of constraints or changes to the objective value must be done through the two interface routines `CPXpreaddrows` and `CPXprechgobj`. We should note that the constraints added by `CPXpreaddrows` are equivalent to but sometimes different from those input by the user. The dual variables associated with the added rows may take different values than those the user might expect.

If a user makes a problem modification that is not consistent with the setting of `CPX_PARAM_REDUCE`, the presolved problem is discarded and presolve is reinvoked at the next optimization call. Similarly, CPLEX discards the presolved problem if the user modifies a variable or constraint that presolve had previously removed from the problem. You can use `CPXpreaddrows` or `CPXprechgobj` to make sure that this will not happen. Note that `CPXpreaddrows` also permits changes to the bounds of the presolved problem. If the nature of the procedure dictates a real need to modify the variables that presolve removed, you can use the `CPXcopyprotected` routine to instruct CPLEX not to remove those variables from the problem.

Instead of changing the bounds on the presolved problem, consider changing the bounds on the original problem. CPLEX will discard the presolved problem, but calling `CPXpresolve` will cause CPLEX to apply presolve to the modified problem, with the added benefit of reductions based on the latest problem modifications. Then use `CPXcrushx`, `CPXcrushpi`,

and `CPXcopystart` to provide an advanced start for the problem after presolve has been applied on the modified problem.

C H A P T E R

27

Parallel Optimizers

This chapter tells you how to use ILOG CPLEX parallel optimizers: Parallel Barrier, Parallel MIP, and Concurrent. All are implemented to run on hardware platforms with parallel processors. These parallel optimizers, like other ILOG CPLEX optimizers, are available in the Interactive Optimizer and in the Component Libraries, if you have an ILOG CPLEX Parallel license. Parallel Barrier, MIP, and Concurrent Optimizers are included with all ILOG CPLEX libraries.

In this chapter, you will learn about:

- *Threads* on page 532
- *Nondeterminism* on page 533
- *Clock Settings and Time Measurement* on page 534
- *Using Parallel Optimizers in the Interactive Optimizer* on page 534
- *Using Parallel Optimizers in the ILOG CPLEX Component Libraries* on page 535
- *Parallel MIP Optimizer* on page 535
- *Parallel Barrier Optimizer* on page 540
- *Concurrent Optimizer* on page 540

Threads

The ILOG CPLEX parallel optimizers are licensed for a specific maximum number of *threads* (that is, the number of processors applied to a given problem). The number of threads that ILOG CPLEX actually uses during a parallel optimization is the *smaller* of:

- the number of threads made available by the operating system;
- the number of threads indicated by the *licensed* values of the thread-limit parameters. Table 27.1 summarizes the values of those thread-limit parameters.

Table 27.1 *Thread-Limit Parameters*

Interactive Command	Concert Technology Enumeration Value	Callable Library Parameter
`set threads`	`Threads`	`CPX_PARAM_THREADS`
`set barrier limits threads`	`BarThreads`	`CPX_PARAM_BARTHREADS`
`set mip limits threads`	`MIPThreads`	`CPX_PARAM_MIPTHREADS`
`set mip limits strongthreads`	`StrongThreadLim`	`CPX_PARAM_STRONGTHREADLIM`

The global thread parameter `Threads` establishes a default thread count for all parallel optimizers. Thread limits for specific optimizers can be set to values that differ from the global default (for example, by setting `IloCplex::MIPThreads`). The default value of the global thread limit is 1 (one). Therefore in order for any of the CPLEX optimizers to invoke parallel threads, the user must do one of the following:

- either the user sets a parameter to a value higher than 1 (one) and lets the CPLEX optimizers determine the way to use the threads; normally, the user sets the global thread limit to establish the default level of parallelism;
- or the user sets any of the *other* thread limits to control the parallelism more explicitly.

The number of threads used when running a parallel CPLEX optimizer is entirely separate from the limit on licensed uses. A typical ILOG CPLEX license permits one licensed use, that is, a single concurrent execution on one licensed computer. If the license also contains the parallel option with a thread limit of, say, four (on a machine with at least four processors), that one concurrent execution of ILOG CPLEX can employ any number of parallel threads to increase performance, up to that limit of 4. A license with the parallel option that additionally has a limit larger than one on the number of licensed uses can support that many simultaneous executions of ILOG CPLEX, each with the licensed maximum number of parallel threads. In such a case, the operating system will manage any contention for processors.

The number of parallel threads used by an ILOG CPLEX optimizer is usually controlled by ILOG CPLEX parameter settings. These settings are discussed in more detail in the sections that follow.

Example: Threads and Licensing

For example, let's assume you use ILOG CPLEX to optimize MIP models on an eight processor machine, and you have purchased an ILOG CPLEX license for four parallel threads. Then you can use the Interactive Optimizer command `set threads i`, substituting values `1` through `4` for `i`. You will not be able to set the thread count higher than `4` because you are licensed for a maximum of four threads.

Threads and Performance Considerations

If you set the number of threads to a value greater than the number of available processors, performance will usually degrade. If you set the number of threads to a value less than the number of processors, the remaining processors will be available for other jobs on your platform. Simultaneously running multiple parallel jobs with a total number of threads exceeding the number of processors may impair the performance of each individual process as its threads compete with one another.

The benefit of applying more threads to optimizing a specific problem varies depending on the optimizer you use and the characteristics of the problem. You should experiment to assess performance improvements and degradation when you apply more or fewer processors. For example, when you optimize the root relaxation using the barrier optimizer, there may be little or no benefit in applying more than four processors to the task. In contrast, if you use 16 processors during the MIP phase of an optimization, you may improve solution speed by a factor of 20. In such a case, you should set the parameters `barrier limit threads` and `mip limit threads` to *different* values in order to use your computing resources efficiently.

Another key consideration in setting optimizer and global thread limits is your management of overall system load.

Nondeterminism

The parallel optimizers are *nondeterministic*: repeated solutions of a model using exactly the same settings can produce different solution paths and, in the case of the parallel MIP optimizer, very different solution times and results.

The basic algorithm in the ILOG CPLEX Parallel MIP Optimizer is branch & cut. The primary source of parallelism in branch & cut is the solution of the continuous relaxations at the individual nodes of the search tree. These subproblems can be distributed over available processors to be carried out in parallel. The individual solution paths for these subproblems

will, in fact, be deterministic, but the speed at which their solutions occur can vary slightly. These variations lead to nodes being taken from and replaced in the branch & cut tree in different order, and this reordering leads to nondeterminism about many other quantities that control the optimization. This nondeterminism is unavoidable in such a context, and its effects can result in some cases in very different solution paths.

The ILOG CPLEX Barrier Optimizer is also not deterministic, but in practice the differences between runs will be minor in comparison to the case of MIP, and should usually amount to a change (if any) of at most a few iterations. The difference is due to uncertainty in the parallel order of arithmetical operations, in conjunction with numerical roundoff.

Clock Settings and Time Measurement

The clock-type parameter determines how ILOG CPLEX measures computation time. CPU time, the default, is most appropriate when only one processor is used. It reports the amount of time the CPU spent performing computation on behalf of your application. For parallel execution, CPU time is system dependent and generally will not reflect the desired metric. On some parallel systems, it may measure aggregate CPU time, that is, the sum of time used by all processors. On others, it may report the CPU time of only one process. In short, it may give you a misleading indication of parallel speed.

The alternative type, wall-clock time, is usually more appropriate for parallel computing because it measures the total physical time elapsed after an operation begins. When multiple processes are active, and when parallel optimizers are active, wall-clock time can be much different from CPU time.

You can choose the type of clock setting, in the:

- Interactive Optimizer, with the command `set clocktype` *`i`*.
- Concert Technology, with the `IloCplex` method `setParam(ClockType,` *`i`*`)`.
- Callable Library, with the routine `CPXsetintparam(env, CPX_PARAM_CLOCKTYPE,` *`i`*`)`.

Replace the *`i`* with the value `1` to specify CPU time or `2` to specify wall-clock time.

Using Parallel Optimizers in the Interactive Optimizer

1. Start the parallel CPLEX Interactive Optimizer with the command `cplex` at the operating system prompt.
2. Set the thread-limit, as explained in *Threads* on page 532.

3. Enter your problem object and populate it as usual.
4. Call the parallel optimizer with the appropriate command from Table 27.2.

Table 27.2 *Parallel Optimizer Commands in the Interactive Optimizer*

Parallel MIP Optimizer	`mipopt`
Parallel Barrier Optimizer	`baropt`
Parallel Concurrent Optimizer	`set lpmethod 6` and then `optimize`

Using Parallel Optimizers in the ILOG CPLEX Component Libraries

1. Create your ILOG CPLEX environment and initialize a problem object in the usual way. See *Initialize the ILOG CPLEX Environment* on page 115 and *Instantiate the Problem Object* on page 116 for details.
2. Within your application, set the appropriate ILOG CPLEX parameter from Table 27.1 to specify the number of threads.
3. Enter and populate your problem object in the usual way, as in *Put Data in the Problem Object* on page 116.
4. Call the parallel optimizer with the appropriate method or routine from Table 27.3.

Table 27.3 *Parallel Optimizer Methods and Routines of Component Libraries*

Optimizer	Concert IloCplex Method	Callable Library
Parallel MIP Optimizer	`solve`	`CPXmipopt`
Parallel Barrier Optimizer	`setParam(RootAlg, Barrier)` and then `solve`	`CPXbaropt` or `CPXhybbaropt`
Concurrent Optimizer	`setParam(RootAlg, Concurrent)` and then `solve`	`CPXsetintparam(env, CPX_PARAM_LPMETHOD, CPX_ALG_CONCURRENT)` and then `CPXlpopt` or `CPXqpopt`

Parallel MIP Optimizer

The ILOG CPLEX Parallel MIP Optimizer delivers significant increases in speed on a wide variety of models, particularly on difficult ones that solve a large number of nodes in the

branch & cut search tree. There are several different opportunities for applying multiple processors to the solution of a MIP problem. These topics highlight those opportunities:

- *Increase the Global Thread Parameter* on page 536
- *Branch & Cut Parallel Processing* on page 536
- *Root Relaxation Parallel Processing* on page 536
- *Individual Optimizer Parallel Processing* on page 537
- *Nested Parallel Processing* on page 537

After those highlights of opportunities to apply multiprocessing, the following sections tell you more about managing parallel MIP optimization:

- *Memory Considerations and the Parallel MIP Optimizer* on page 537
- *Output from the Parallel MIP Optimizer* on page 538

Increase the Global Thread Parameter

The most straightforward way to invoke parallelism is by setting the global thread parameter, `Threads`, to a value greater than `1` to indicate the desired degree of parallelism. If the other parameters remain set to their default values, the result is that nodes in the branch & cut tree will be processed in parallel; that is, each node is solved in its entirety on one of the available processors, independently of the work being performed on other processors. In typical cases, the number of nodes waiting for solution quickly becomes greater than the number of threads, creating enough work which can be done in parallel to make speed increases possible. ILOG CPLEX automatically manages the pool of nodes, so that each time a thread finishes one node, it is assigned to solving another.

Branch & Cut Parallel Processing

A contrasting and specialized approach to obtaining speed increases by parallel processing within the branch & cut tree is to:

1. choose strong branching (`VarSel` parameter setting `3`) as the variable selection strategy;
2. apply multiple processors to the strong branching variable selection computation by setting the strong branching thread limit, `StrongThreadLim`, to a value greater than `1`; and
3. leaving the global thread limit at `1` to inhibit processing the branching and solution of the nodes in parallel.

On models where strong branching is already a beneficial technique, this approach to parallelism can be especially attractive.

Root Relaxation Parallel Processing

In some models, the continuous root relaxation takes significant solution time before parallel branching begins. These models have potential for additional speed increases by running the root relaxation step in parallel. If the root problem is an LP and the `Threads` parameter is

set to a value greater than 1, the concurrent optimizer is invoked by default. This provides a form of parallelism that applies the available threads to multiple optimizers. If the root problem is a QCP, the dual simplex optimizer alone is used.

You can try a different form of parallelism at the root by selecting a specific optimizer with the starting algorithm parameter (RootAlg in Concert, CPX_PARAM_STARTALG in the Callable Library, set mip strategy startalgorithm in the Interactive Optimizer) to select the Barrier optimizer. The parallel threads will all be applied to the barrier algorithm.

Individual Optimizer Parallel Processing

Parallelism in barrier is ordinarily controlled by the global thread count parameter, but this default behavior can be overridden by the individual optimizer thread limit parameter, BarThreads. The degree of parallelism within the branch & cut tree likewise can be controlled by the MIP thread limit MipThreads, which overrides the global thread limit. This capability to set either or both MipThreads and BarThreads permits great flexibility in the use of parallel resources during the solution of a MIP model.

For example, on a model where only a small number of nodes is active at any one time, the benefit of parallel solution of these nodes is limited. If the individual nodes are computationally challenging, then it may be possible to achieve speed increases by leaving the global thread limit at its default of 1 and setting the continuous optimizer thread limit (BarThreads) to a value greater than 1. The global thread limit of 1 will inhibit the parallelism of the branching process, while the explicit thread limit of more than 1 will permit the optimization of each node in parallel.

Nested Parallel Processing

Nested parallelism represents a further way to exploit the flexibility of independent thread parameters. For example, it might be determined from experience in a given family of models that only a modest degree of parallelism is beneficial at the nodes and additional processors do not help speed up the branching. In such a case, better speed increases might be obtained by combining a parallelization of the work that the continuous optimizer does at each node. On an 8-processor computer, you might opt to solve a model by setting the MipThreads limit to 4 instead of its maximum of 8, and the BarThreads limit to 2, thus keeping all 8 processors busy as much of the time as possible, with the four MIP threads each invoking two threads for the barrier optimizer.

If you do decide to try a nested parallel approach, keep in mind the rule of thumb that it is usually better to keep a higher degree of parallelism of the nodes themselves (MipThreads) than of the continuous optimizer (BarThreads); this is in keeping with the general observation that MIP speed increases are on average closer to linear in the number of threads than the speed increases for the continuous optimizers.

Memory Considerations and the Parallel MIP Optimizer

Before the parallel MIP optimizer invokes parallel processing, it makes separate, internal copies of the initial problem. The individual processors use these copies during computation,

so each of them requires an amount of memory roughly equal to the size of the original model after it is presolved.

Output from the Parallel MIP Optimizer

The parallel MIP optimizer generates slightly different output in the node log (see *Progress Reports: Interpreting the Node Log* on page 331) from the serial MIP optimizer. The following paragraphs explain those differences.

If the MIP optimizer is running in parallel, it will display elapsed time for the MIP optimizer in wall-clock time, independent of the setting of the clock-type parameter (assuming MIP logging has not been turned off).

ILOG CPLEX prints a summary of timing statistics specific to the parallel MIP optimizer at the end of optimization. You can see typical timing statistics in the following sample run.

```
Problem 'fixnet6.mps.gz' read.
Read time =    0.02 sec.
CPLEX> o
Tried aggregator 1 time.
MIP Presolve modified 308 coefficients.
Aggregator did 1 substitutions.
Reduced MIP has 477 rows, 877 columns, and 1754 nonzeros.
Presolve time =    0.02 sec.
Clique table members: 2
MIP emphasis: balance optimality and feasibility
Root relaxation solution time =    0.07 sec.

        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer     Best Node    ItCnt     Gap

      0     0     3192.0420    12                  3192.0420        8
*     0+    0                   0     4505.0000     3192.0420        8   29.14%
                  3384.5860    15     4505.0000     Cuts:  36       51   24.87%
                  3513.7923    17     4505.0000     Cuts:  25       92   22.00%
                  3530.1967    19     4505.0000  Flowcuts:  9      104   21.64%
*     0+    0                   0     4471.0000     3530.1967       104   21.04%
                  3604.4590    17     4471.0000 Flowcuts:  10      124   19.38%
                  3607.9420    18     4471.0000  Flowcuts:  4      131   19.30%
*     0+    0                   0     4448.0000     3607.9420       131   18.89%
                  3608.7548    20     4448.0000  Flowcuts:  3      136   18.87%
                  3617.6257    19     4448.0000  Flowcuts:  2      141   18.67%
                  3624.7454    19     4448.0000  Flowcuts:  2      150   18.51%
                  3627.2428    20     4448.0000  Flowcuts:  1      152   18.45%
*     0+    0                   0     3994.0000     3627.2428       152    9.18%
*    30+    5                   0     3985.0000     3736.8034       305    6.23%
*    46     0                   0     3983.0000     3972.7760       326    0.26%

Root relaxation processing (before b&c):
  CPU      time            =    0.60
Parallel b&c, 2 threads:
```

```
  Real     time             =    0.72
  Critical time (total)     =    0.00
  Spin     time (average)   =    0.01
                              -------
Total (sequential+parallel) =    1.32 sec.

Cover cuts applied:  1
Flow cuts applied:  38
Gomory fractional cuts applied: 9

Integer optimal solution:  Objective =    3.9830000000e+03
Solution time =    0.90 sec.  Iterations = 328  Nodes = 47

CPLEX> q
```

The summary at the end of the sample says that 0.60 of a second was spent in the phase of processing the root relaxation, that is, all the combined steps (preprocessing, root relaxation solution, cutting planes, heuristic) that occur at the root before the first branch occurs. The parallel part of this sample run took 0.72 of a second of real time (that is, elapsed time for that phase).

Other parts of the sample report indicate that the processors spent an average of 0.01 of a second of real time spinning (that is, waiting for work while there were too few active nodes available). The real critical time was a total of 0.00 seconds, time spent by individual processors in updating global, shared information. Since only one processor can access the critical region at any instant in time, the amount of time spent in this region really is crucial: any other processor that tries to access this region must wait, thus sitting idle, and this idle time is counted separately from the spin time.

There is another difference in the way logging occurs in the parallel MIP optimizer. When this optimizer is called, it makes a number of copies of the problem. These copies are known as *clones*. The parallel MIP optimizer creates as many clones as there are threads available to it. When the optimizer exits, these clones and the memory they used are discarded.

If a log file is active when the clones are created, then ILOG CPLEX creates a clone log file for each clone. The clone log files are named `cloneK.log`, where `K` is the index of the clone, ranging from 0 (zero) to the number of threads minus one. Since the clones are created at each call to the parallel MIP optimizer and discarded when it exits, the clone logs are opened at each call and closed at each exit. (The clone log files are not removed when the clones themselves are discarded.)

The clone logs contain information normally recorded in the ordinary log file (by default, `cplex.log`) but inconvenient to send through the normal log channel. The information likely to be of most interest to you are special messages, such as error messages, that result from calls to the LP optimizers called for the subproblems.

Parallel Barrier Optimizer

The ILOG CPLEX Parallel Barrier Optimizer achieves significant speedups over its serial counterpart on a wide variety of classes of problems. (The serial Barrier Optimizer is introduced in Chapter 9, *Solving LPs: Barrier Optimizer*, and explored further in Chapter 11, *Solving Problems with a Quadratic Objective (QP)* and in Chapter 12, *Solving Problems with Quadratic Constraints (QCP)*.) Consequently, the parallel barrier optimizer will be the best continuous choice on a parallel computer more frequently than on a single-processor. For that reason, you should be careful *not* to apply performance data or experience based on *serial* optimizers when you are choosing which optimizer to use on a parallel platform.

If you decide to use the parallel barrier optimizer on the *subproblems* of a MIP, see also other special considerations about nested parallelism in *Nested Parallel Processing* on page 537.

Concurrent Optimizer

On a multiprocessor computer, the concurrent optimizer launches distinct LP and QP optimizers on multiple threads, terminating as soon as the first optimizer finishes. The first thread uses the same strategy as the single-processor `automatic LPMethod` setting `(0)`. If a second thread is available, the concurrent optimizer runs the barrier optimizer on it. If a third processor is available, dual simplex, primal simplex, and barrier are all run. All further available threads are devoted to making the barrier optimization parallel. It should be noted that the barrier optimization is not considered complete until the crossover step has been performed and simplex re-optimization has converged; in other words, regardless of which optimizer turns out to be the fastest, the concurrent optimizer always returns a basic solution at optimality.

The concurrent optimizer requires more memory than any individual optimizer, and of course it adds system load by consuming more aggregate CPU time than the fastest individual optimizer would alone. But the advantages offered in terms of robust solution of models, and assurance in most cases of the minimum solution time, will make it attractive in many situations.

Index

A

absolute objective difference **314**, **338**
absolute optimality tolerance **338**
  gap **338**
accessing
  basis information **56**
  current parameter value **52**, **128**
  default parameter value **128**
  dual values **56**
  maximum parameter value **128**
  minimum parameter value **128**
  objective function value (C++) **56**
  reduced costs **56**
  solution quality **57**
  solution values **55**
active model **79**
  MIP **84**
active node **454**
`add` method
  `IloModel` class **47**, **58**
`addMinimize` **81**
advanced basis
  example **215**
  in networks **186**
  parallel threads and **185**
  primal feasibility and **185**
  reading from file (LP) **190**
  saving to file (LP) **190**
  starting from (LP) **190**
advanced starting basis **215**
`AggCutLim` **320**
aggregator
  barrier preprocessing **236**
  simplex and **188**
`alg` **85**
algorithm
  choosing in LP **51**
  controlling in `IloCplex` **52**
  pricing **191**
  type **85**
  using multiple **382**
`Algorithm.Barrier` **89**
application
  creating with Concert Technology **43**
  development steps **43**
arc **246**
architecture **113**
arguments
  null pointers **122**
  optional **122**
array
  constructing **92**
  creating multi-dimensional **63**
  extensible **47**
  using for I/O **64**

B

barrier **89**

barrier optimizer
algorithm **226**
algorithms and infeasibility **243**
barrier display parameter **240**
centering corrections **240**
column nonzeros parameter **237**, **241**
corrections limit **240**
growth parameter **242**
infeasibility analysis **243**
inhibiting dual formulation **188**
linear **225** to **243**
log file **230**
numerical difficulties and **241**
out-of-core barrier **236**
parallel **540**
performance tuning **235**
preprocessing **236**
primal-dual **186**
quadratic **265** to **293**
row-ordering algorithms **238**
simplex optimizer and **227**
solution quality **233**
solving LP problems **225**
starting-point heuristics **238**
unbounded optimal face and **242**
unbounded problems **242**
uses **226**
working storage allocation **236**
BarThreads **537**
BAS file format **199**
basis **89**
accessing information **56**
advanced, primal feasible **185**
advanced, role in network **186**
advanced, starting from (example) **215**
column generation and **376**
condition number **198**, **203**
crash parameter and **194**
crossover algorithms **227**
current **115**
differences between LP and network optimizers **259**
from basis file **259**
infeasibility and **201**
maximum row residuals and **204**
no factoring in network optimizer **258**
objective in Phase I and **201**
optimal, condition number and **198**
optimal, numerical stability and **198**
parallel threads and advanced basis **185**
preprocessing versus starting (MIP) **323**
previous optimal **117**
refactoring rate **193**
removing objects from **59**
role in converting LP to network flow **259**
role in converting network-flow to LP **259**
role in network optimizer **249**
role in network optimizer to solve QP **250**
saving best so far **199**
singularities and **199**
starting from advanced **522**
unexplored nodes in tree (MIP) **340**
unstable optimal **203**
see also advanced basis
Best Bound **316**
Best Estimate **316**
bibliography **36**
column generation **376**
models for LPs, MIPs **390**
Boolean or semi-continuous variables
single LP **83**
`boolVar` **77**
`boolVarArray` **77**
bound
violation **203**
`Bounded` return status (Java) **81**
branch & cut **84**, **515**
algorithm **313**
controlling process with presolve **523**
parallel processing **536**
parameters **315**
solves MIP models **311**
tree **313**
branch & cut algorithm
memory problems and **340**
priority order **324**
special ordered sets (SOS) **329**
storing tree nodes **341**
tree subproblems **344**
branch & cut tree
solving linear problem at each node **522**

branch variable selection callback **517**
branching directions **87**
breakpoint
  discontinuous piecewise linear and **399**
  example **398**
  piecewise linear function and **398**
`BtTol` **315**

C

call by value **118**
Callable Library
  categories of routines **114**
  core **114**
  debugging and **150**
  description **26**
  parameters **127**
  using **113** to **141**
  see also individual `CPX`*xxx* routines
callback
  branch variable selection **517**
  control **496**
  cut **516**
  diagnostic **478**
  graphic user interface and **489**
  heuristic **515**
  incumbent **518**
  node selection **519**
  resetting to null **53**, **129**
  solve **519**
  using status variables **488**
cardinality as logical constraint **414**
changing
  coefficients in model **380**
  limit on barrier corrections **240**
  maximization to minimization **252**
  pricing algorithm **249**
  problem type
    network to LP **259**
    qp **271**
    zeroed_qp **271**
  quadratic coefficients **270**
  type of variable **381**
  variable type **59**
channel
  example **164**
character string
  length requirements **123**
`check.c` CPLEX file **124**
Cholesky factor **235**
  barrier iteration **226**
  barrier log file and **232**
choosing
  algorithm in LP **51**
clique cuts
  defined **318**
`cloneK.log` **539**
clones **539**
  log files **539**
  threads and **539**
closing
  application **117**, **252**
  environment **117**, **252**
  log files **161**
coefficient, changing **380**
column
  dense **241**
  density **237**
  growth parameter **252**
  index number **122**
  modeling by **92**
  name **122**
  nonzeros parameter **237**, **241**
  referencing **122**
column generation
  basis and **376**
  cutting plane method and **376**
  reduced cost and **376**, **378**
columnwise modeling **62**, **75**, **92**, **130**
comma-separated values *see* CSV file format
complementarity **226**, **242**
  convergence tolerance **241**
Component Libraries
  defined **26**
concave
  logical constraints and **416**
  piecewise linear **401**
Concert Technology
  accessing parameter values **52**
  application development steps **43**

creating application **43**
description **26**
design **41**
error handling **60**
solving problem with **41**
using **41** to **71**
writing programs with **41**
see also individual `Ilo`*xxx* methods
Concurrent Optimizer **540**
concurrent optimizer **540**
licensing issues **532**
non-default parameters and **187**
option for **52**
parallel processing and **537**
root relaxation and **537**
constraint
adding with user-written callback **516**
convex quadratic **295**
cumulative **207**, **209**
cuts as **318**
lazy **507**, **516**
logical **411**
modeling linear **49**
quadratic **295**
removing from basis **59**
representing with `IloRange` **46**
violation **204**
constraints
create ranged **75**
ranged **78**
constructing arrays of variables **92**
continuous piecewise linear **399**
continuous relaxation **84**, **313**
continuous relaxation subproblem **454**
control callback
definition **496**
types of **496**
conventions
character strings **123**
naming **158**
notation **33**
numbering **153**, **156**
numbering rows, columns **153**
convergence tolerance
barrier algorithm and **187**
definitioin **235**
effect of tightness **241**
performance and **235**
`convert` CPLEX utility **159**
converting
error code to string **251**
file formats **159**
network-flow model to LP **258**
network-flow problem to LP **259**
convex
logical constraints and **416**
piecewise linear **401**
convex quadratic constraint **295**
convex quadratic programming *see* quadratic programming

cover cuts **318**
defined **318**
CPLEX
Component Libraries **26**
core **114**
licensing **42**
parameters **52**
`cplex` object
creating **49**
notifying changes to **58**
`cplex.h` header file **125**, **127**, **128**, **152**
`cplex.log` file
changing name **230**
clone logs **539**
default name **161**, **192**
`CPX_ALG_AUTO` symbolic constant **345**
`CPX_ALG_CONCURRENT` symbolic constant **345**
`CPX_ALG_DUAL` symbolic constant **345**
`CPX_ALG_HYBNETOPT` symbolic constant **345**
`CPX_ALG_PRIMAL` symbolic constant **345**
`CPX_ALG_SIFTING` symbolic constant **345**
`CPX_INTEGER_INFEASIBLE` **518**
`CPX_PARAM_AGGIND` parameter **323**
`CPX_PARAM_BARSTARTALG` parameter **239**
`CPX_PARAM_BARTHREADS` parameter **532**
`CPX_PARAM_BBINTERVAL` parameter **315**
`CPX_PARAM_BNDSTRENIND` parameter **323**
`CPX_PARAM_BRDIR` parameter **315**
`CPX_PARAM_BTTOL` parameter **315**
`CPX_PARAM_CLIQUES` parameter **320**

CPX_PARAM_CLOCKTYPE parameter **128**, **534**
CPX_PARAM_COEREDIND parameter **323**
CPX_PARAM_COLGROWTH parameter **120**, **252**
CPX_PARAM_COVERS parameter **320**
CPX_PARAM_DISJCUTS parameter **320**
CPX_PARAM_FLOWCOVERS parameter **320**
CPX_PARAM_FLOWPATHS parameter **320**
CPX_PARAM_FRACCUTS parameter **320**
CPX_PARAM_GUBCOVERS parameter **320**
CPX_PARAM_IMPLBD parameter **320**
CPX_PARAM_INTSOLLIM parameter **326**
CPX_PARAM_MIPCBREDLP **515**, **517**, **518**
CPX_PARAM_MIPTHREADS parameter **532**
CPX_PARAM_MIRCUTS parameter **320**
CPX_PARAM_NODELIM parameter **326**
CPX_PARAM_NODESEL parameter **315**
CPX_PARAM_NZGROWTH parameter **120**
CPX_PARAM_PREIND parameter **323**
CPX_PARAM_PRELINEAR **516**, **525**
CPX_PARAM_PREPASS parameter **323**
CPX_PARAM_PROBE parameter **322**
CPX_PARAM_REDUCE **516**, **524**, **525**, **526**, **528**
CPX_PARAM_REDUCE parameter **323**
CPX_PARAM_RELAXPREIND **522**
CPX_PARAM_RELAXPREIND parameter **323**
CPX_PARAM_ROWGROWTH parameter **120**, **252**
CPX_PARAM_SCRIND parameter **124**, **152**, **164**, **199**, **218**, **251**, **488**
CPX_PARAM_STARTALG parameter **344**
CPX_PARAM_STRONGTHREADLIM parameter **532**
CPX_PARAM_THREADS parameter **532**
CPX_PARAM_TILIM parameter **326**
CPX_PARAM_TRELIM parameter **326**
CPX_PARAM_VARSEL parameter **315**
CPX_PARAM_WORKMEM parameter **326**
CPX_PREREDUCE_DUALONLY **524**
CPX_PREREDUCE_NO_PRIMALORDUAL **524**
CPX_PREREDUCE_PRIMALANDDUAL **524**
CPX_PREREDUCE_PRIMALONLY **524**
CPX_SEMICONT **331**
CPX_SEMIINT **331**
CPXaddchannel routine **119**, **164**
CPXaddcols **528**
CPXaddcols routine **120**, **146**
CPXaddfpdest routine **125**, **164**, **165**
CPXaddfuncdest routine **125**, **164**, **165**
CPXaddrows routine **120**, **131**, **146**
CPXaddusercuts **525**
CPXALG_BARRIER symbolic constant **345**
CPXbaropt **526**
CPXbasicpresolve **528**
CPXCENVptr **119**
CPXCHANNELptr data type **119**
CPXCHARptr data type **125**
CPXcheckaddcols routine **124**
CPXcheckaddrows routine **124**
CPXcheckchgcoeflist routine **124**
CPXcheckcopyctype routine **124**
CPXcheckcopylp routine **124**
CPXcheckcopylpwnames routine **124**
CPXcheckcopyqsep routine **124**
CPXcheckcopyquad routine **124**
CPXcheckcopysos routine **124**
CPXcheckvals routine **124**
CPXchgbds **528**
CPXchgcoeflist routine **146**
CPXchgprobtype routine **272**, **348**
CPXchgqpcoef routine **270**, **271**
CPXchgrhs **528**
CPXcloneprob **516**, **527**
CPXcloseCPLEX routine **117**, **165**, **218**, **252**, **278**, **286**, **358**, **488**
CPXCLPptr **119**
CPXCNETptr **119**
CPXcopybase **526**
CPXcopybase routine **218**
CPXcopycttype routine **331**
CPXcopyctype routine **151**, **307**, **347**
CPXcopylp routine **116**, **146**
CPXcopynettolp routine **258**
CPXcopyorder routine **330**, **364**
CPXcopyprotected **518**, **525**
CPXcopyquad routine **278**
CPXcopysos routine **330**, **331**, **364**
CPXcopystart **526**, **527**
CPXcreateprob **488**
CPXcreateprob routine **116**, **119**, **131**, **218**, **278**, **286**, **347**, **358**
CPXcutcallbackadd **516**, **525**
CPXdelchannel routine **164**, **165**

CPXdelfpdest routine **125**, **164**, **165**
CPXdelfuncdest routine **164**, **165**
CPXdisconnectchannel routine **164**
CPXdisplayiis routine **205**
CPXdualopt **519**, **526**
CPXENVptr data type **119**
CPXERR_NEGATIVE_SURPLUS symbolic constant **442**
CPXERR_PRESLV_INF **525**
CPXERR_PRESLV_UNBD **525**
CPXERR_PRESOLVE_BAD_PARAM **508**
cpxerror message channel **163**, **165**
CPXfclose routine **125**
CPXFILEptr data type **125**
CPXflushchannel routine **164**
CPXfopen routine **125**, **161**
CPXfputs routine **126**
CPXfreepresolve **525**
CPXfreeprob **525**
CPXfreeprob routine **117**, **218**, **278**, **286**, **358**, **488**
CPXgetcallbackgloballb **515**, **516**
CPXgetcallbackglobalub **515**
CPXgetcallbackincumbent **515**
CPXgetcallbackinfo routine **125**, **486**, **487**, **488**
CPXgetcallbacklp **515**, **516**
CPXgetcallbacknodeintfeas **517**, **518**
CPXgetcallbacknodelb **515**
CPXgetcallbacknodelp **515**
CPXgetcallbacknodeub **515**
CPXgetcallbacknodex **516**
CPXgetcallbackorder **517**
CPXgetcallbackpseudocosts **517**
CPXgetchannels routine **119**, **163**, **165**
CPXgetcolindex routine **123**
CPXgetcolname routine **443**
CPXgetcols routine **441**, **442**, **443**
CPXgetctype routine **308**
CPXgetdblparam routine **122**, **128**
CPXgetdblquality routine **199**, **204**, **233**
CPXgeterrorstring routine **251**, **488**
CPXgetintparam routine **122**, **128**
CPXgetintquality routine **233**
CPXgetmipobjval routine **348**, **358**
CPXgetmipslack routine **348**
CPXgetmipx routine **348**, **358**
CPXgetnumcols routine **121**
CPXgetredlp **527**
CPXgetrowindex routine **123**
CPXgetrowname routine **121**
CPXgetsos routine **308**
CPXgetstat routine **348**, **358**, **487**
CPXgetstrparam routine **122**, **128**
CPXgetx routine **118**, **119**
CPXhybnetopt **519**
CPXiiswrite routine **205**, **260**
CPXinfodblparam routine **122**, **128**
CPXinfointparam routine **122**, **128**
CPXinfostrparam routine **122**, **128**
cpxlog message channel **163**
CPXlpopt **278**, **286**
CPXlpopt routine **131**, **489**
CPXLPptr data type **119**
CPXmemcpy routine **126**
CPXMIP_ABORT_FEAS symbolic constant **487**
CPXMIP_ABORT_INFEAS symbolic constant **487**
CPXmipopt **526**
CPXmipopt routine **348**, **357**
CPXmsg routine **115**, **125**, **164**, **165**
CPXmsgstr routine **126**
CPXNETaddarcs routine **251**
CPXNETaddnodes routine **251**
CPXNETcheckcopynet routine **124**
CPXNETchgobjsen routine **252**
CPXNETcreateprob routine **119**, **251**
CPXNETdelnodes routine **252**
CPXNETfreeprob routine **252**
CPXNETprimopt routine **252**, **260**
CPXNETptr data type **119**
CPXNETsolution routine **252**
CPXnewcols routine **131**, **146**
CPXnewrows routine **146**
CPXopenCPLEX routine **115**, **119**, **128**, **131**, **163**, **165**, **218**, **251**, **278**, **286**
CPXordwrite routine **364**
CPXpreaddrows **528**
CPXpresolve **526**, **527**
CPXprimopt **526**
CPXprimopt routine **127**, **348**
CPXPROB_FIXEDMILP symbolic constant **348**
CPXPUBLIC symbolic constant **125**
CPXPUBVARARGS symbolic constant **125**

CPXqpopt routine **278**, **286**
CPXqpwrite routine **269**
CPXreadcopyprob routine **116**, **444**
cpxresults message channel **163**
CPXsavwrite routine **150**
CPXsetbranchcallbackfunc **517**
CPXsetcutcallbackfunc **516**
CPXsetdblparam routine **122**, **128**
CPXsetdefaults routine **129**
CPXsetheuristiccallbackfunc **515**
CPXsetintparam routine **122**, **128**, **152**, **218**, **251**, **344**, **534**
CPXsetlogfile routine **124**, **125**, **152**, **161**, **163**, **230**
CPXsetlpcallbackfunc routine **125**, **487**, **489**
CPXsetmipcallbackfunc routine **125**, **487**
CPXsetnodecallbackfunc **519**
CPXsetsolvecallbackfunc **519**
CPXsetstrparam routine **122**, **129**
CPXsolution routine **131**, **348**, **487**
CPXsoswrite routine **364**
CPXstrcpy routine **126**
CPXstrlen routine **126**
CPXtreewrite routine **326**
CPXVOIDptr data type **125**
cpxwarning message channel **163**
CPXwriteprob routine **131**, **150**, **199**, **364**
CPXwritesol routine **123**
create
  modeling variables **75**
  new rows **91**
  objective function **75**
  ranged constraints **75**
creating
  application with Concert Technology **43**
  CPLEX environment **251**
  log file **161**
  network flow problem object **251**
  problem object **116**
crossover
  file format **230**
  irreducibly inconsistent set (IIS) and **243**
  verifying barrier solutions **239**
CSV file format **160**
cumulative constraint **207**, **209**
cut
  dual reductions and **525**
cut callback **516**
cuts **318**, **454**
  adding **319**
  clique **318**
  cover **318**
  disjunctive **318**
  flow cover **318**
  flow path **319**
  Gomory fractional **319**
  GUB cover **319**
  implied bound **319**
  local or global **455**
  MIR **319**
  recorded in MIP node log file **333**
  re-optimizing **319**
  what are **318**
cutting plane method **376**

D

data
  entering **116**
data types
  special **119**
debugging
  Callable Library and **150**
  diagnostic routines and **124**
  heap **153**
  Interactive Optimizer and **150**
  return values and **152**
definition **306**
degeneracy **200**
  dual **344**
delete goal stacks **461**
deleting
  model objects **58**
deleting nodes **461**
deleting variables **98**
dense column **237**
Depth First **316**
destroying
  CPLEX environment **117**
  nodes **252**
  problem object **117**

devex pricing **193**
diagnosing
  infeasibility in QPs **273**
  network infeasibility **260**
diagnostic callback
  definition **478**
  types of **478**
diagnostic routine **124**
  log file and **124**
  message channels and **124**
  redirecting output from **124**
diet model **79**
`diff` **77**
dimensions, checking **153**
discontinuous piecewise linear **399**
  breakpoints and **399**
  segments and **399**
disjunctive cuts **318**
displaying
  barrier information **230**
  barrier solution quality **233**
  basis condition **199**
  bound infeasibilities **203**
  column-nonzeros parameter **241**
  infeasibilities on screen **203**, **205**
  messages **164**
  MIP information periodically **334**
  network objective values **248**
  network solution information **251**
  network solution on screen **252**
  optimization progress **489**
  problem dimensions **153**
  problem statistics **153**
  reduced-cost infeasibilities **203**
  simplex solution quality **235**
  solution quality **202**
  solutions on screen **165**
  variables **156**
DPE file format **344**
dual feasibility **226**
dual reduction **201**
dual residual **203**
dual simplex optimizer
  perturbing objective function **200**
  selecting **185**
  stalling **200**
  see also simplex
dual variable
  solution data **56**
duality gap **226**

E

elapsed time for the MIP optimizer **538**
empty goal **456**, **461**
`end` method
  `IloEnv` class **44**
`enter` Interactive Optimizer command **307**
entering **307**
  LPs for barrier optimizer **228**
  mixed integer programs (MIPs) **307**
  network arcs **251**
  network data **251**
  network data from file **260**
  network nodes **251**
enumeration
  `Algorithm` **51**
  `BasisStatus` **56**
  `BoolParam` **52**
  `IntParam` **52**
  `NumParam` **52**
  `Quality` **58**
  `Status` **54**
  `String Param` **52**
environment
  initializing **115**
  multithreaded **115**
  releasing **117**
environment object
  constructing **44**
`EpAGap` **338**
`EpGap` **338**
`eq` method **78**
error checking
  diagnostic routines for **124**
  MPS file format **158**
  problem dimensions **153**
error handling
  in Concert Technology **60**
  querying exceptions **152**

Error return status (C++) **55**
Error return status (Java) **81**
example
  Column Generation **375**
  columnwise modeling **62**, **130**
  contents of IIS file **206**
  creating multi-dimensional arrays **63**
  Cutting Stock **375**
  FORTRAN **126**
  message handler **164**
  MIP node log file **332**
  MIP optimization **346**
  MIP problem from file **356**
  MIP with SOS and priority orders **362**
  network optimization **247**
  optimizing QP **273**
  output channels **164**
  output from infeasibility finder **206**
  Piecewise Linear **397**
  reading QP from file **278**, **286**
  rowwise modeling **62**, **130**
  using arrays for I/O **64**
executing a goal **454**
expr **78**
expression
  building **45**
  editable **77**
  linear **45**
  logical **45**
  piecewise linear **45**
  using modeling variables to construct **75**
external variables **118**
extra rim vectors **157**

F

FailGoal **455**
feasibility
  analysis and barrier optimizer **243**
  check **455**
  dual **194**, **226**
  network flows and **247**
  primal **226**
  progress toward **200**, **247**
feasibility tolerance
  default **203**
  largest bound violation and **203**
  network optimizer and **249**
  range of **203**
  reducing **202**
Feasible return status (C++) **55**
Feasible return status (Java) **81**
file format
  converting **159**
  described **156**
  example of quadratic program **278**, **286**
file reading routines in Callable Library **114**
file writing routines in Callable Library **114**
flow cover cuts
  defined **318**
flow path cuts
  defined **319**
FORTRAN **126**, **153**
FracCand **321**
FracPass **321**
fractional cuts
  defined **319**
free row **157**

G

ge method **78**
generalized upper bound (GUB)
  see also GUB
generalized upper bound (GUB) cover cuts **319**
getBasisStatus method
  IloCplex Java class **89**
getBasisStatuses method
  IloCplex C++ class **56**
getBoundSA **89**
getBoundSA method
  IloCplex class **56**
getCplexStatus method
  IloCplex class **55**, **57**
getDefault method
  IloCplex class **52**
getDual **89**
getDual method
  IloCplex class **56**
getDuals **89**

getDuals method
IloCplex class **56**
getIIS **90**
getIIS method
IloCplex class **57**
getMax **87**
getMax method
IloCplex class **52**
getMessage method
IloException class **152**
getMin **87**
getMin method
IloCplex class **52**
getObjSA **89**
getObjSA method
IloCplex class **56**
getObjValue method
IloCplex class **56**
getParam **87**
getParam method
IloCplex class **52**
getQuality method
IloCplex class **57**, **199**, **233**
getRangeSA **89**
getReducedCost **89**
getReducedCost method
IloCplex class **56**
getReducedCosts **89**
getReducedCosts method
IloCplex class **56**
getRHSSA method
IloCplex class **56**
getSlack method
IloCplex class **56**
getSlacks method
IloCplex class **56**
getStatus method
IloCplex class **54**
IloCplex::Exception class **152**
getStatuses method
IloCplex class **216**
getting *see* accessing
getValue method
IloCplex class **55**
getValues method
IloCplex class **56**
global thread limit **537**
global thread parameter **536**
global variables **118**
goal
empty **456**
executing **454**
goal stack **460**
goals as parameters **455**
Gomory fractional cuts
defined **319**
graphic user interface (GUI) **489**
GUB
constraint **319**

H

head **246**
header file **152**
heap, debugging **153**
heuristic
definition **321**
node **321**
relaxation induced neighborhood search (RINS) **321**
RINSHeur **321**, **322**
SubMIPNodeLimMDefault Para Font> **321**, **322**
heuristic callback **515**
heuristic solutions **468**
heuristics
starting point **238**

I

ill-conditioned
basis **204**
factors in **204**
maximum dual residual and **204**
problem **202**
ILM *see* ILOG License Manager (ILM)
IloAdd template class **65**
IloAddable **75**, **79**
IloAlgorithm::Exception class **60**
IloAlgorithm::Status enumeration **54**
IloArray template class **47**
IloColumn class

and example **94**
IloColumnArray **92**
IloConstraint class **49**
IloConversion **98**
IloConversion class **44**, **49**, **59**
IloCplex **75**
implements IloMPModeler **75**
objects in user application **42**
IloCplex class
getBasisStatus method **89**
getBasisStatuses method **56**
getBoundSA method **56**
getCplexStatus method **55**, **57**
getDefault method **52**
getDual method **56**
getDuals method **56**
getIIS method **57**
getMax method **52**
getMin method **52**
getObjSA method **56**
getObjValue method **56**
getParam method **52**
getQuality method **57**, **199**, **233**
getReducedCost method **56**
getReducedCosts method **56**
getRHSSA method **56**
getSlack method **56**
getSlacks method **56**
getStatus method **54**
getStatuses method **216**
getValue method **55**
getValues method **56**
isDualFeasible method **55**
isPrimalFeasible method **55**
setDefaults method **53**
setParam method **53**
solve method **50**, **54**, **55**, **57**, **58**, **59**, **66**
writeBasis method **199**
IloCplex::Algorithm enumeration **51**
IloCplex::BarStartAlg **239**
IloCplex::BasisStatus enumeration **56**
IloCplex::BoolParam enumeration **52**
IloCplex::Exception class **60**, **152**
getStatus method **152**
IloCplex::IntParam enumeration **52**
IloCplex::Kappa **199**
IloCplex::NumParam enumeration **52**
IloCplex::Quality enumeration **58**
IloCplex::Reduce **202**
IloCplex::StringParam enumeration **52**
IloCPModeler **75**
IloEnv class **44**
end method **44**
IloException class
getMessage method **152**
IloExpr class **45**
ILOG License Manager (ILM)
CPLEX and **42**
IloMaximize function **46**
IloMinimize function **46**, **65**
IloModel class **46**
add method **47**, **58**
remove method **47**, **58**
IloModel.add **98**
IloModel.delete **98**
IloModel.remove **98**
IloModeler **75**, **76**
basic modeling **76**
IloMPModeler **75**, **76**, **91**
IloNumArray class **47**
IloNumExpr **77**, **78**
IloNumVar **75**
extension of IloNumExpr **77**
IloNumVar class **44**, **49**
IloNumVarArray class **45**
IloObjective **79**, **92**
IloObjective class **45**, **49**, **65**
setExpr method **270**
IloObjectiveSense **78**
IloObjectiveSense.Maximize **78**
IloObjectiveSense.Minimize **78**
iloqpex1.cpp example
example
iloqpex1.cpp **274**
IloRange **79**, **92**
IloRange class **46**, **49**
IloSemiContVar class **49**
IloSolver **74**
IloSOS1 class **49**
IloSOS2 class **49**

implied bound cuts
  defined **319**
include file **152**
inconsistent constraints **205**
incumbent
  node **314**
  solution **314**
incumbent callback **518**
index number **122**
Individual optimizer parallel processing **537**
`inf.getNumVar` **90**
`inf.getRange` **90**
infeasibility
  barrier optimizer and **242**
  diagnosing in network flows **260**
  diagnosing in QPs **273**
  displaying on screen **203**
  dual **240**, **243**
  maximum bound **203**
  maximum reduced-cost **203**
  network flow **247**
  network optimizer and **260**
  norms **232**
  primal **233**, **240**, **243**
  ratio in barrier log file **233**
  reports **200**
  scaling and **202**
  unscaled **202**
infeasibility finder **204**, **243**
  multiple IISs **205**
  network-flows and **260**
  preprocessing and **209**
  sample output **205**
  setting time limit on **208**
infeasible problem
  analyzing **57**
`Infeasible` return status (C++) **55**
`Infeasible` return status (Java) **82**
infeasible solution
  information **89**
`InfeasibleOrUnbounded`
  return status **82**
`InfeasibleOrUnbounded` return status (C++) **55**
initializing
  CPLEX environment **251**
  problem object **116**, **251**
input operator **47**
instantiating
  CPLEX environment **251**
  problem object **116**, **251**
`int` **86**
integer variables
  single LP **83**
integrality constraints **454**
integrality tolerance
  MIP **339**
  parameter **339**
Interactive Optimizer
  debugging and **150**
  description **26**
  experimenting with optimizers **148**
  improving application performance **150**
  testing code in **146**
`IntParam` **85**
`IntSolLim` **326**
`intVar` **77**
irreducibly inconsistent constraints **205**
irreducibly inconsistent set (IIS)
  algorithms **209**
  example, cumulative constraint **207**
  example, network **260**
  Java API and **90**
  network flows and **260**
  sample file **206**
`isDualFeasible` method
  `IloCplex` class **55**
`isPrimalFeasible` method
  `IloCplex` class **55**

K

knapsack constraint **318**, **319**
knapsack with reduced cost in objective **376**

L

lazy constraint **516**
  definition **507**
  pool **507** to **511**
lazy constraints

in pools **507**
`le` method **78**
license
CPLEX **42**
parallel **531**
limiting
network iterations **249**
strong branching candidate list **338**
strong branching iterations **338**
linear expression **45**
linear objective function **49**
Linear Programming problem *see* LP
linear relaxation
network flow and **260**
log file
barrier optimizer **230**
Cholesky factor in **232**
clones and **539**
closing **161**
contents **192**, **233**
creating **161**
default **161**
description **161**
diagnostic routines and **124**
iteration **196**
naming **161**
network **249**
parallel MIP optimizer and **539**
parameter **161**
records infeasibilities **203**
records infeasibility information **205**
records singularities **199**
relocating **161**
renaming **161**
logical constraint **411**, **412**
example in early tardy scheduling **429**
logical expression **45**
LP
barrier optimizer **225**
choosing algorithm **51**
network optimizer **245**
problem formulation **26**, **226**
solving **183** to ??, **183** to **243**
LP file format
IISs and **206**
QPs and **268**
row, column order **156**
special considerations **156**

M

managing
log file **161**
Markowitz tolerance **199**, **200**
default **200**
increasing to stay feasible **200**
maximum value **199**
numerical stability and **199**
pivots and **199**
slow progress in Phase I and **200**
maximal cliques
recorded in MIP node log file **333**
maximization
concave QPs **266**
lower cutoff parameter **339**
see also optimization
`maximize` **78**
maximum bound infeasibility **203**
maximum reduced-cost infeasibility **203**
maximum row residual **204**
memory **340**
memory allocation **236**
memory leaks **44**
memory management **195**, **340**
refactoring frequency and **196**
message channel
diagnostic routines and **124**
message handler (example) **164**
MILP
definition **84**
minimal covers
recorded in MIP node log file **333**
minimization
convex QPs **266**
see also optimization
upper cutoff parameter **339**
`minimize` **78**
MIP **305** to **374**
active model **84**
branch & cut **84**

changing variable type **310**
memory problems and **340**
optimizer **305**
problem formulation **306**
progress reports **331**
relaxation algorithm **344**
subproblem algorithm **344**
subproblems **343**
supplying first integer solution **324**
terminating optimization **325**
`MIPEmphasis` **311**
`MipThreads` **537**
MIQP
definition **84**
MIR
cuts **319**
Mixed Integer Linear Program (MILP)
definition **306**
Mixed Integer Programming (MIP)
definition **306**
Mixed Integer Programming problem *see* MIP
Mixed Integer Quadratic Program (MIQP)
definition **306**
Mixed Integer Quadratically Constrained Program (MIQCP) **306**
model
active **79**
adding columns to **380**
adding objects **58**
adding submodels **46**
changing coefficients in **380**
changing variable type **381**
consistency and tolerance **499**
deleting objects **58**
extracting **50**
manipulations **75**
modifying **98**
notifying changes to `cplex` object **58**
removing objects **58**
serializing **159**
solving **42**, **50**
solving with `IloCplex` **66**
XML representation of **159**
modeling
columnwise **62**, **130**
objects **42**
rowwise **62**, **130**
modeling by column **75**, **92**
modeling variable
`IloNumVar` **76**
modelling variables
create **75**
modifying a model **98**
MPS file format
CPLEX extensions **157**
reference row values **331**
saving modifications **158**
saving QP **268**
weights **331**
multithreaded application
needs multiple environments **115**

N

`name` **78**
namespace conflicts **118**
naming
arcs in network flow **251**
conventions **158**
log file **161**
node file **342**
nodes in network flow **251**
`negative` **77**
negative semi-definite
objective **267**
nested parallel processing **537**
NET file format **252**
`NetItLim` **249**
network
converting to LP model **258**
embedded **249**
infeasibility in **247**
modeling variables **246**
problem formulation **246**, **247**
network extractor **250**
Network Flow problem
see network
network object **246**
network optimizer **186**, **245** to **251**
column growth parameter **252**

preprocessing and **251**
problem formulation **247**
row growth parameter **252**
turn off preprocessing **251**
`nexample.net` **248**
node **454**
demand **247**
from **246**
head **246**
sink **247**
solving linear problem at **522**
source **247**
supply **247**
tail **246**
to **246**
transshipment **247**
viable **515**
node file **341**
`cpx` name convention **342**
using with MIP **318**
when to use **341**
node heuristic **321**
node log **331**
node problem **454**
node selection callback **519**
node selection strategy
best estimate **343**
depth-first search **343**
`NodeLim` **326**
nondeterminism **533**
nonlinear expression
definition **413**
nonseparable **266**
notation **33**
notation in this manual **33**
notifying
changes to `cplex` object **58**
numbering conventions
C **153**
Fortran **153**
row, column order **156**
numerical difficulties **196**, **241**
barrier growth parameter **242**
basis condition number and **198**
complementarity **241**
convergence tolerance **241**
dense columns removed **241**
infeasibility and **200**
sensitivity **198**
unbounded optimal face **241**
numerical variable **49**
`numVarArray` **77**

O

`ObjDif` tolerance parameter **314**
objective
optimality tolerance and **204**
objective coefficients
crash parameter and **194**
modified in log file **248**
network flows and **248**
priority and **325**
objective difference
absolute **314**, **338**
relative **314**, **338**
objective function **78**
accessing value of (C++) **56**
changing sense **252**
constructor **78**
create **75**
free row as **157**
in log file **248**
in MPS file format **158**
maximization **158**
maximize **46**
minimize **46**
network flows and **246**
preprocessing and **201**
primal reductions and **201**
quadratic terms in a single LP **84**
representing with `IloObjective` **45**
sign reversal in **158**
objective value
accessing slack in (C++) **56**
in log file **248**
network flows and **246**
object range parameter **242**
unbounded **242**
operator << **47**

operator >> **47**
`Optimal` return status (C++) **55**
`Optimal` return status (Java) **82**
optimality
  basis condition number and **198**
  cutoff parameters **339**
  duality and **344**
  infeasibility ration **233**
  normalized error and **235**
  singularities and **199**
  tolerance **202**, **203**
    relative **338**
optimality tolerance
  absolute **338**
  changing relative or absolute **338**
  gap **338**
  maximum reduced-cost infeasibility and **203**
  Network and **249**
  reducing **202**
  relative **338**
  relative, default **338**
  setting **204**
  when to change **338**
optimization
  interrupting **489**
  stopping **325**, **489**
optimization problem
  defining with modeling objects **42**
  representing with `IloModel` **46**
optimization routines in Callable Library **114**
optimizer
  barrier (linear) **225** to **243**
  barrier (quadratic) **265** to **293**
  concurrent **540**
  differences between Barrier, simplex **227**
  dual simplex **185**
  MIP **305**
  network **186**, **245** to **251**
  parallel **531** to **540**
  primal simplex **186**
  primal-dual barrier **186**
  selecting **83**
optimizing
  cuts **319**
ORD file format
  setting priorities in **330**
OrGoal **460**
out-of-core barrier **236**
output
  channel parameter **162**
  debugging and **152**
  redirecting **162**
output operator **47**

P

parallel
  license **531**, **532**
  optimizers **531** to **540**
  threads **532**
Parallel Barrier Optimizer **540**
Parallel MIP Optimizer **535**
  memory considerations **537**
  output log file **538**
parallel processing
  branch & cut **536**
  individual optimizer **537**
  nested **537**
  root relaxation **536**
  selected starting algorithm **537**
parameter
  accessing current value **52**, **128**
  accessing default value **128**
  accessing maximum value **128**
  accessing minimum value **128**
  algorithmic **235**
  barrier corrections **240**
  Callable Library and **127**
  controlling branch & cut strategy **315**
  gradient **191**
  iisfind **209**
  log file **161**
  netfind **250**
  object range **242**
  optimality cutoff **339**
  output channel **162**
  perturbation limit **200**
  preprocessing dependency **237**
  screen indicator **251**
  setting **128**

setting all defaults **53**, **129**
setting in C++ **53**
symbolic constants as **128**
tree memory **340**
parameter routines in Callable Library **114**
Parameters **86**
parameters
max and min values for **87**
query **87**
performance
convergence tolerance and **235**
perturbing
objective function **200**
variable bounds **200**
piecewise linear **397**
breakpoint **399**
concave **401**
continuous **399**
convex **401**
definition **397**
discontinuous **399**
example **397**
example in early tardy scheduling **429**
steps **399**
piecewise linear expression **45**
piecewise linear functions **75**
single LP **84**
pool
of cuts **507**
of lazy constraints **507**
of user cuts **507**
populating problem object **116**, **251**
portability **125**
positive semi-definite
objective **267**
quadratic constraint **298**
PreCompress **196**
preprocessing
advanced basis and (LP) **191**
barrier and **236**
barrier optimizer **236**
dense columns removed **241**
dependency parameter **237**
dual reductions in **201**
IIS files and **209**
infeasibility finder and **209**
MIPs **322**
network optimizer and **251**
primal reductions in **201**
simplex **188**
simplex and **188**
starting-point heuristics and **239**
presolve **86**
dependency checking in **188**
final factorization after uncrush in **189**
gathering information about **527**
interface **526**
limited **527**
process **521**
process for MIP **522**
protecting variables during **525**
restricting dual reductions **524**
restricting primal reductions **524**
presolve compression parameter **189**
presolved problem
adding constraints to **523**
and branch & cut process **514**
building **522**
freeing **525**
freeing memory **528**
retaining **525**
presolver
barrier preprocessing **236**
simplex and **188**
simplex preprocessing **188**
pricing algorithms **249**
primal feasibility **226**
primal reduction
reduction
primal **201**
primal simplex algorithm **87**
primal simplex optimizer **186**
perturbing variable bounds **200**
stalling **200**
see also simplex
primal variables **194**
primal-degenerate problem **185**
primal-dual barrier optimizer
see barrier optimizer
`PrimalPricing` class **86**

priority **325**
  binary variables and **325**
  derived from set members **330**
  integer variables and **325**
  order **324**, **330**
  parameter to control **325**
  reading from file **325**
  semi-continuous variables and **325**
  semi-integer variables and **325**
  special ordered set (SOS) and **325**
  special ordered sets (SOSs) and **330**
priority orders **87**
probing parameter **322**
problem
  analyzing infeasible **57**
  solving with Concert Technology **41**
problem description
  example: Rates **392**
  example: semi-continuous variables **392**
  example:Column Generation **376**
  example:Cutting Stock **376**
problem formulation
  barrier **226**
  dual **226**, **228**
  ill-conditioned **202**
  infeasibility reports **200**
  linear **26**
  network **247**
  network-flow **246**
  primal **226**, **228**
  removing dense columns **237**
  switching from network to LP **258**, **259**
problem modification routines in Callable Library **114**
problem object
  creating **116**
  destroying **117**
  freeing **117**
  initializing **116**
  instantiating **116**
  network **246**
  populating **116**, **251**
problem query routines in Callable Library **114**
problem representation
  example: Rates **392**
  example: semi-continuous variables **392**
  example:Column Generation **378**
  example:Cutting Stock **378**
problem solution
  example: Rates **393**
  example: semi-continuous variables **393**
  example:Column Generation **382**
  example:Cutting Stock **382**
problem type
  changing from network to LP **259**
  changing to qp **271**
  changing to zeroed_qp **271**
  quadratic programming and **269**
`prod` **77**
pruned node **454**
PSD
  positive semi-definite in objective function **267**

Q

QP
  example **273**, **278**, **286**
  problem formulation **266**
  solution example **278**, **286**
  solving **265** to **293**
QP file format **269**
QP relaxation **272**
quadratic
  constraints **295**
  convex constraints **295**
quadratic coefficient
  changing **270**
quadratic objective functions **49**
Quadratic Programming problem *see* QP
quadratically constrained programming (QCP) **295** to ??, **295**, ?? to **301**
query routine **443**

R

ranged constraints **78**
  create **75**
ranged row **157**
reading
  MIP problem data **356**
  MIP problem data from file **308**

MIP tree from TRE file **326**
network data from file **260**
perturbed problem from DPE file **344**
problem data from VEC file **230**
QP problem data from file **278**, **286**
start values from MST file **324**
redirecting
diagnostic routines **124**
log file output **162**
oputput **152**
screen output **162**
reduced cost
column generation and **376**, **378**
reduced costs **89**
accessing **56**
reduced-cost pricing **193**
reduction
dual **201**
refactoring frequency
dual simplex algorithm and **187**
primal simplex algorithm and **187**
reference counting **462**
reference row values **331**
reflection scaling **250**
relative objective difference **314**, **338**
relative optimality tolerance **338**
relaxation
algorithm applied to **344**
continuous **84**
of MIP problem **313**
QP **272**
series of **84**
relaxation induced neighborhood search (RINS) **321**
`RelObjDif` tolerance parameter **314**
relocating
log file **161**
`remove` method
`IloModel` class **47**, **58**
renaming
log file **161**
residual
dual **203**
maximum dual **204**
maximum row **204**
row **203**
return status
`Bounded` (Java) **81**
`Error` (C++) **55**
`Error` (Java) **81**
`Feasible` (C++) **55**
`Feasible` (Java) **81**
`Infeasible` (C++) **55**
`Infeasible` (Java) **82**
`InfeasibleOrUnbounded` **82**
`InfeasibleOrUnbounded` (C++) **55**
`Optimal` (C++) **55**
`Optimal` (Java) **82**
`Unbounded` **82**
`Unbounded` (C++) **55**
`Unknown` (C++) **55**
`Unknown` (Java) **81**
return value **121**
debugging with **152**
routines to access parameters **128**
RHS *see* right-hand side
right-hand side (RHS)
file formats for **157**
rim vectors **157**
`RINSHeur` **321**, **322**
root relaxation
parallel processing **536**
`RootAlg` **85**
row
growth parameter **252**
index number **122**
name **122**
referencing **122**
residual **203**
row-ordering algorithms **238**
approximate minimum degree (AMD) **238**
approximate minimum fill (AMF) **238**
automatic **238**
nested dissection (ND) **238**
rowwise modeling **62**, **130**

S

SAV file format **269**
saving
best factorable basis **199**

DPE file **344**
MIP tree **326**
perturbed problem **344**
SAV file **344**
TRE file **327**
scaling **202**, **204**
alternative methods of **193**
definition **193**
in network extraction **250**
infeasibility and **202**
numeric difficulties and QP **273**
objective function in QP **273**
singularities and **199**
search tree **454**
semi-continuous variable **49**, **331**, **392**
priority and **325**
semi-continuous variables **75**
semi-definite
negative and objective **267**
positive and constraints **298**
positive and objective **267**
semi-integer variable **391**
priority and **325**
sensitivity analysis **89**
performing **56**
separable **266**
serializing **159**
`setDefaults` **87**
`setDefaults` method
`IloCplex` class **53**
`setDirection` **88**
`setDirections` **88**
`setExpr` method
`IloObjective` class **270**
`setLB` **98**
`setLinearCoef` **98**
setOut **161**
`setParam` **85**, **86**
`setParam` method
`IloCplex` class **53**
`setPriorities` **88**
`setPriority` **88**
setting
all default parameters **53**, **129**
callbacks to null **53**, **129**
parameters **128**
parameters in C++ **53**
see also changing
`setUB` **98**
sifting **186**
simplex
dual **185**
feasibility tolerance in MIP **339**
optimizer **227**
primal **186**
see also dual simplex optimizer
see also primal simplex optimizer
simplex method
column generation and **375**, **376**
pricing phase and **375**, **376**
singularity **199**
slack
accessing bound violations in (C++) **57**
accessing in constraints in active model (Java) **83**
accessing slack variables in constraints (C++) **56**
accessing slack variables in objective (C++) **56**
as indicator of ill-conditioning **204**
as reduced cost in infeasibility analysis **204**
example `CPXgetmipslack` **348**
in primal formulation (Barrier) **226**
in summary statistics **203**
infeasibilities as bound violations and **204**
infeasibility in dual associated with reduced costs **203**
initial norms and **192**
maximum bound violation and (Java) **90**
pivoted in when constraint is removed **59**
primal bound error in solution quality (Barrier) **234**
reducing computation of steepest edge pricing **193**
role in inequality constraints (Barrier) **232**
role in infeasibility analysis **204**
row complementarity in solution quality (Barrier) **234**
steepest edge and (dual) **192**
steepest edge and (primal) **192**
using primal variables instead **194**
variable needed in basis (Network) **259**
variables and primal variables (dual) **194**
solution
accessing quality information **57**
accessing values of **55**
basic infeasible primal **201**

basis **227**
complementary **226**
differences between barrier, simplex **227**
example QP **278**, **286**
feasible in MIPs **324**
incumbent **314**
infeasible basis **243**
midface **227**
nonbasis **227**
quality **233**, **239**
serializing **159**
supplying first integer in MIPs **324**
using advanced presolved **526**
verifying **239**
XML representation of **159**
solve callback **519**
`solve` method
`IloCplex` class **50**, **54**, **55**, **57**, **58**, **59**, **66**
solving
diet problem (Java example) **83**
model **50**
single LP (Java) **85**
subsequent LPs or QPs in a MIP (Java) **85**
SOS
branching priority **330**
role in model (Java) **75**
setting priorities and **330**
type 1 **49**
type 2 **49**
using **329**
weights in **331**
sparse matrix **91**
Special Ordered Set *see* SOS
special ordered sets
single LP **83**
speed increase **536**
`square` **77**
stalling **200**
starting algorithm
parallel processing **537**
static variables **118**
status variables, using **488**
steepest-edge pricing **193**, **341**
step in piecewise linear function **399**
`string` **86**
`StringParam` **86**
strong branching **338**
`StrongThreadLim` **536**
`SubMIPNodeLimMDefault Para Font>` **321**, **322**
`sum` **77**
surplus argument **442**
symbolic constants **121**, **128**

T

tail **246**
terminating
because of singularities **199**
because of stalling **200**
because of time limit **209**
MIP optimization **325**
network optimizer iterations **249**
threads **532**
clones **539**
parallel optimizers **532**
performance and **533**
`Threads` global parameter **536**
thread-safe **118**
`TiLim` **326**
time limit
concurrent optimizer and **187**
effects all algorithms invoked by concurrent optimizer **187**
Interactive Optimizer command (example) **209**
possible reason for `Unknown` return status (C++) **55**
possible reason for `Unknown` return status (Java) **81**
setting for infeasibility finder (example) **208**
`TiLim` parameter (MIP) **312**
tolerance
absolute objective difference and **314**
absolute optimality **338**
advice about **499**
complementarity convergence, default of **241**
complementary solution and **226**
consistency in model and **499**
convergence and barrier algorithm **187**
convergence and numeric inconsistencies **241**
convergence and performance **235**
cut callbacks and **499**
cut callbacks and (example) **499**

cuts in goals and **467**
default numerical (example LP) **197**
feaibility range **203**
feasibility (Network) **249**
feasibility and largest bound violation **203**
feasibility default **203**
feasibility, reducing **202**
integrality (Java example) **98**
Markowitz **199**
Markowitz and numerical difficulty **200**
Markowitz, increasing to stay feasible **200**
optimality **203**
optimality (Network) **249**
optimality, reducing **202**
relative objective difference and **314**
relative optimality **338**
relative optimality default **338**
role of (C++) **57**
role of (Java) **90**
simplex feasibility in cut callback **499**
simplex optimality (example C++) **53**
singularities and scaling **199**
termination and **325**
violated constraints in goals and **467**
warning about absolute and relative objective difference **314**
when reducing does not help **203**
TRE file format **326**, **341**
`TreLim` **326**, **341**
type
changing variable (Concert) **75**
conversion (Java) **98**

U

unbounded optimal face **227**, **241**
`Unbounded` return status **82**
`Unbounded` return status (C++) **55**
`Unknown` return status (C++) **55**
`Unknown` return status (Java) **81**
user cut
definition **507**
pool **507** to **511**
user cuts
in pools **507**
utility routines in Callable Library **114**

V

variable
accessing dual **56**
cannot be added to a model **79**
changing type **44**, **59**
changing type of **381**
constructing an array **92**
deleting **98**
external **118**
global **118**
in expressions **45**
modeling **76**
numerical **49**
order **156**, **157**
removing from basis **59**
representing with `IloNumVar` **44**
semi-continuous **49**, **331**, **392**
semi-integer **391**
static **118**
type **307**
variable selection strategy
strong branching **337**, **343**
variable type change **75**
variables
create modeling **75**
semi-continuous **75**
`VarSel` **536**
VEC file format **230**
vectors, rim **157**
violation
bound **203**
constraint **204**

W

`WorkDir` **236**, **342**
working memory **236**
`WorkMem` **236**, **326**, **340**
`writeBasis` method
`IloCplex` class **199**

X

XML
Concert Technology and **159**
serializing model, solution **159**