



Changing the rules of business™

ILOG CPLEX 10.2

User's Manual

March 2007

COPYRIGHT NOTICE

ILOG CPLEX 10.2 Copyright © 1997-2007, by ILOG S.A., 9 Rue de Verdun, 94253 Gentilly Cedex, France, and ILOG, Inc., 1080 Linda Vista Ave., Mountain View, California 94043, USA. All rights reserved.

General Use Restrictions

This document and the software described in this document are the property of ILOG and are protected as ILOG trade secrets. They are furnished under a license or nondisclosure agreement, and may be used or copied only within the terms of such license or nondisclosure agreement.

No part of this work may be reproduced or disseminated in any form or by any means, without the prior written permission of ILOG S.A, or ILOG, Inc.

Trademarks

ILOG, the ILOG design, CPLEX, and all other logos and product and service names of ILOG are registered trademarks or trademarks of ILOG in France, the U.S. and/or other countries.

All other company and product names are trademarks or registered trademarks of their respective holders.

Java and all Java-based marks are either trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Microsoft and Windows are either trademarks or registered trademarks of Microsoft Corporation in the United States and other countries.

document version 10.2

Table of Contents

Preface	Meet ILOG CPLEX.....	25
Part I	Languages and APIs	39
Chapter 1	ILOG Concert Technology for C++ Users.....	41
	Architecture of a CPLEX C++ Application.....	42
	Licenses	42
	Compiling and Linking	43
	Creating a C++ Application with Concert Technology	43
	Modeling an Optimization Problem with Concert Technology	43
	Modeling Classes.....	44
	Creating the Environment: IloEnv.....	44
	Defining Variables and Expressions: IloNumVar	44
	Declaring the Objective: IloObjective	45
	Adding Constraints: IloConstraint and IloRange	46
	Formulating a Problem: IloModel	46
	Data Management Classes	47
	Solving the Model	48
	Extracting a Model	49

Invoking a Solver	50
Choosing an Optimizer.	51
Controlling the Optimizers	53
Accessing Solution Information	55
Accessing Solution Status	55
Querying Solution Data	56
Accessing Basis Information	57
Performing Sensitivity Analysis	57
Analyzing Infeasible Problems.	57
Solution Quality	58
Modifying a Model	59
Deleting and Removing Modeling Objects	59
Changing Variable Type.	60
Handling Errors	61
Example: Optimizing the Diet Problem in C++	63
Problem Representation	63
Creating a Model Row by Row.	63
Creating a Model Column by Column	64
Application Description.	65
Creating Multi-Dimensional Arrays with IloArray	65
Using Arrays for Input/Output.	65
Building the Model by Row.	66
Building the Model by Column	66
Solving the Model with IloCplex.	67
Complete Program	68
Chapter 2 ILOG Concert Technology for Java Users	69
Architecture of a CPLEX Java Application	70
Licenses.	71
Compiling and Linking	71
Creating a Java Application with Concert Technology	71
Modeling an Optimization Problem with Concert Technology	72

Using IloModeler	74
Modeling Variables	74
Building Expressions	74
Ranged Constraints	75
Objective Functions	76
The Active Model	76
Building the Model	77
Solving the Model	79
Accessing Solution Information	80
Choosing an Optimizer	81
Solving a Single Continuous Model	82
Solving Subsequent Continuous Relaxations in a MIP	83
Controlling ILOG CPLEX Optimizers	84
Parameters	84
Priority Orders and Branching Directions	86
More Solution Information	86
Writing Solution Files	86
Dual Solution Information	87
Basis Information	87
Infeasible Solution Information	88
Solution Quality	88
Advanced Modeling with IloLPMatrix	89
Modeling by Column	90
Example: Optimizing the Diet Problem in Java	91
Modifying the Model	92
Chapter 3 ILOG Concert Technology for .NET Users	95
Describe	96
What is known?	97
What are the unknowns?	97
What are the constraints?	97
What is the objective?	97

	Model	98
	Build by Rows	99
	Build by Columns	101
	Solve	102
	Good Programming Practices	103
	Example: Optimizing the Diet Problem in C#.NET	105
Chapter 4	ILOG CPLEX Callable Library	107
	Architecture of the ILOG CPLEX Callable Library	108
	Licenses	109
	Compiling and Linking	109
	Using the Callable Library in an Application	109
	Initialize the ILOG CPLEX Environment	109
	Instantiate the Problem Object	110
	Put Data in the Problem Object	110
	Optimize the Problem	111
	Change the Problem Object	111
	Destroy the Problem Object	112
	Release the ILOG CPLEX Environment	112
	ILOG CPLEX Programming Practices	112
	Variable Names and Calling Conventions	112
	Data Types	114
	Ownership of Problem Data	114
	Problem Size and Memory Allocation Issues	114
	Status and Return Values	115
	Symbolic Constants	115
	Parameter Routines	116
	Null Arguments	116
	Row and Column References	116
	Character Strings	117
	Checking Problem Data	117
	Using the Data Checking Parameter	117

Using Diagnostic Routines for Debugging	118
Callbacks	118
Portability	119
CPXPUBLIC	119
Function Pointers	119
CPXCHARptr, CPXCCHARptr, and CPXVOIDptr	119
File Pointers	119
String Functions	120
FORTTRAN Interface	120
Case-Sensitivity	120
Underscore	120
Six-Character Identifiers	120
Call by Reference	121
Pointers	121
Strings	121
C++ Interface	121
Managing Parameters from the Callable Library	121
Example: Optimizing the Diet Problem in the Callable Library	123
Problem Representation	123
Creating a Model Row by Row	124
Creating a Model Column by Column	124
Program Description	125
Solving the Model with CPXlpopt	125
Using Surplus Arguments for Array Allocations	126
Example: Using Query Routines lpex7.c	127

Part II Programming Considerations 129

Chapter 5 Developing CPLEX Applications	131
Tips for Successful Application Development	131
Prototype the Model	132
Identify Routines to Use	132

Test Interactively	132
Assemble Data Efficiently	132
Test Data	133
Choose an Optimizer	133
Program with a View toward Maintenance and Modifications	134
Comment Your Code	134
Write Readable Code	134
Avoid Side-Effects	135
Don't Change Argument Values.	135
Declare the Type of Return Values	135
Manage the Flow of Your Code	135
Localize Variables	135
Name Your Constants	135
Choose Clarity First, Efficiency Later	136
Debug Effectively	136
Test Correctness, Test Performance	136
Using the Interactive Optimizer for Debugging.	136
Eliminating Common Programming Errors.	138
Check Your Include Files	139
Clean House and Try Again.	139
Read Your Messages.	139
Check Return Values	139
Beware of Numbering Conventions	139
Make Local Variables Temporarily Global	140
Solve the Problem You Intended	140
Special Considerations for Fortran.	140
Tell Us	140
Chapter 6 Managing Input and Output.	141
Understanding File Formats	142
Working with LP Files.	142
Variable Order and LP Files	142

Working with MPS Files	143
Free Rows in MPS Files	143
Ranged Rows in MPS Files	143
Extra Rim Vectors in MPS Files	143
Naming Conventions in MPS Files	144
Error Checking in MPS Files	144
Saving Modified MPS Files	144
Converting File Formats	144
Using Concert XML Extensions	145
Using Concert csvReader	146
Managing Log Files	147
Creating, Renaming, Relocating Log Files	147
Closing Log Files	147
Controlling Message Channels	148
Parameter for Output Channels	148
Callable Library Routines for Message Channels	149
Example: Callable Library Message Channels	150
Concert Technology Message Channels	152
Chapter 7 Licensing an Application	153
Types of ILM Runtime Licenses	154
File-Based RTNODE, RTSTOKEN or TOKEN Keys	154
Memory-Based RUNTIME Keys	154
Routines and Methods Used for Licensing	154
Examples	155
CPXputenv Routine for C and C++ Users	155
The putenv Method for Java Users	156
The Putenv Method for .NET Users	157
CPXRegisterLicense Routine for C and C++ Users	157
The registerLicense Method for Java Users	157
The RegisterLicense Method for .NET Users	158
Summary	158

Part III Continuous Optimization. 159

Chapter 8 Solving LPs: Simplex Optimizers 161

Choosing an Optimizer for Your LP Problem 162

Automatic Selection of Optimizer 163

Dual Simplex Optimizer 163

Primal Simplex Optimizer 164

Network Optimizer 164

Barrier Optimizer 164

Sifting Optimizer 164

Concurrent Optimizer 165

Parameter Settings and Optimizer Choice 165

Tuning LP Performance. 165

Preprocessing 166

 Dual Formulation in Presolve 166

 Dependency Checking in Presolve 166

 Final Factor after Presolve 167

 Memory Use and Presolve 167

 Controlling Passes in Preprocessing 167

 Aggregator Fill in Preprocessing 168

 Turning Off Preprocessing 168

Starting from an Advanced Basis 168

Simplex Parameters 169

 Pricing Algorithm and Gradient Parameters 169

 Scaling 171

 Refactoring Frequency 171

 Crash 172

 Memory Management and Problem Growth 172

Diagnosing Performance Problems 173

Lack of Memory 173

 Warning Messages 173

Paging Virtual Memory	174
Refactoring Frequency and Memory Requirements.	174
Preprocessing and Memory Requirements	174
Numeric Difficulties	174
Numerical Emphasis Settings	175
Numerically Sensitive Data	175
Measuring Problem Sensitivity with Basis Condition Number	177
Repeated Singularities	177
Stalling Due to Degeneracy	178
Inability to Stay Feasible	179
Diagnosing LP Infeasibility	179
Coping with an Ill-Conditioned Problem or Handling Unscaled Infeasibilities	180
Interpreting Solution Quality.	181
Maximum Bound Infeasibility: Identifying Largest Bound Violation	182
Maximum Reduced-Cost Infeasibility	182
Maximum Row Residual.	182
Maximum Dual Residual	182
Maximum Absolute Values: Detecting Ill-Conditioned Problems	183
Finding a Conflict	183
Repairing Infeasibility: FeasOpt	183
Example: Using a Starting Basis in an LP Problem	184
Example ilolpex6.cpp	184
Example lpex6.c.	184
Chapter 9 Solving LPs: Barrier Optimizer	187
Introducing the Barrier Optimizer.	188
Barrier Simplex Crossover	189
Differences between Barrier and Simplex Optimizers	189
Using the Barrier Optimizer	190
Special Options	191
Controlling Crossover	191
Using SOL File Format	192

	Interpreting the Barrier Log File	192
	Nonzeros in Lower Triangle of AA^T in the Log File	193
	Ordering-Algorithm Time in the Log File	194
	Cholesky Factor in the Log File	194
	Iteration Progress in the Log File	194
	Infeasibility Ratio in the Log File	195
	Understanding Solution Quality from the Barrier LP Optimizer	195
	Tuning Barrier Optimizer Performance	197
	Memory Emphasis: Letting the Optimizer Use Disk for Storage	198
	Preprocessing	199
	Detecting and Eliminating Dense Columns	200
	Choosing an Ordering Algorithm	200
	Using a Starting-Point Heuristic	201
	Overcoming Numeric Difficulties	201
	Numerical Emphasis Settings	202
	Difficulties in the Quality of Solution	202
	Change the Barrier Algorithm	203
	Change the Limit on Barrier Corrections	203
	Choose a Different Starting-Point Heuristic	203
	Difficulties during Optimization	204
	Difficulties with Unbounded Problems	205
	Diagnosing Infeasibility Reported by Barrier Optimizer	206
Chapter 10	Solving Network-Flow Problems	207
	Choosing an Optimizer: Network Considerations	208
	Formulating a Network Problem	208
	Example: Network Optimizer in the Interactive Optimizer	209
	Understanding the Network Log File	210
	Tuning Performance of the Network Optimizer	211
	Controlling Tolerance	211
	Selecting a Pricing Algorithm for the Network Optimizer	211
	Limiting Iterations in the Network Optimizer	211

	Solving Problems with the Network Optimizer	211
	Network Extraction	212
	Preprocessing and the Network Optimizer	213
	Example: Using the Network Optimizer with the Callable Library netex1.c	213
	Solving Network-Flow Problems as LP Problems	214
	Example: Network to LP Transformation netex2.c	216
Chapter 11	Solving Problems with a Quadratic Objective (QP)	217
	Identifying Convex QPs	218
	Entering QPs	219
	Matrix View	219
	Algebraic View	220
	Examples for Entering QPs	220
	Reformulating QPs to Save Memory	221
	Saving QP Problems	222
	Changing Problem Type in QPs	222
	Changing Quadratic Terms	223
	Optimizing QPs	224
	Diagnosing QP Infeasibility	225
	Example: Creating a QP, Optimizing, Finding a Solution	226
	Example: ilqpex1.cpp	226
	Example: QPex1.java	227
	Example: qpex1.c	227
	Example: Reading a QP from a File qpex2.c	228
Chapter 12	Solving Problems with Quadratic Constraints (QCP)	229
	Identifying a Quadratically Constrained Program (QCP)	229
	Convexity	230
	Semi-definiteness	232
	Second Order Cone Programming (SOCP)	232
	Determining Problem Type	233
	Concert Technology and QCP Problem Type	233

Callable Library and QCP Problem Type	233
Interactive Optimizer and QCP Problem Type	233
File Formats and QCP Problem Type	233
Changing Problem Type	239
Changing Quadratic Constraints	240
Solving with Quadratic Constraints	240
Numeric Difficulties and Quadratic Constraints	241
Examples: QCP	241

Part IV Discrete Optimization 243

Chapter 13 Solving Mixed Integer Programming Problems (MIP)	245
Stating a MIP Problem	246
Considering Preliminary Issues	247
Entering MIP Problems	247
Displaying MIP Problems	248
Changing Problem Type in MIPs	249
Changing Variable Type.	250
Using the Mixed Integer Optimizer	251
Emphasizing Feasibility and Optimality	251
Terminating MIP Optimization	253
Tuning Performance Features of the Mixed Integer Optimizer	254
Branch & Cut	255
Applying Cutoff Values	256
Applying Tolerance Parameters	256
Applying Heuristics	256
When an Integer Solution Is Found: the Incumbent	256
Controlling Strategies: Diving and Backtracking	257
Selecting Nodes	258
Selecting Variables	259
Changing Branching Direction	259
Solving Subproblems	260

Using Node Files	260
Probing	260
Cuts	261
Clique Cuts	262
Cover Cuts	262
Disjunctive Cuts	262
Flow Cover Cuts	262
Flow Path Cuts	262
Gomory Fractional Cuts	262
Generalized Upper Bound (GUB) Cover Cuts	263
Implied Bound Cuts	263
Mixed Integer Rounding (MIR) Cuts	263
Adding Cuts and Re-Optimizing	263
Counting Cuts	263
Parameters Affecting Cuts	264
Heuristics	265
Node Heuristic	265
Relaxation Induced Neighborhood Search (RINS) Heuristic	265
Solution Polishing	266
Preprocessing: Presolver and Aggregator	267
Starting from a Solution	269
Issuing Priority Orders	270
Using the MIP Solution	271
Progress Reports: Interpreting the Node Log	273
Troubleshooting MIP Performance Problems	277
Too Much Time at Node 0	278
Trouble Finding More than One Feasible Solution	278
Large Number of Unhelpful Cuts	279
Lack of Movement in the Best Node	279
Time Wasted on Overly Tight Optimality Criteria	279
Slightly Infeasible Integer Variables	280

	Running out of Memory	281
	Reset the Tree Memory Parameter	282
	Use Node Files for Storage	282
	Change Algorithms	285
	Difficulty Solving Subproblems: Overcoming Degeneracy	285
	Unsatisfactory Subproblem Optimization	286
	RootAlg Parameter	286
	NodeAlg Parameter	287
	Examples: Optimizing a Basic MIP Problem	288
	ilomipex1.cpp	288
	mipex1.c	288
	Example: Reading a MIP Problem from a File	289
	ilomipex2.cpp	289
	mipex2.c	289
Chapter 14	Using Special Ordered Sets (SOS)	291
	What Is a Special Ordered Set (SOS)?	291
	Example: SOS Type 1 for Sizing a Warehouse	292
	Declaring SOS Members	293
	Examples: Using SOS and Priority	293
	ilomipex3.cpp	293
	mipex3.c	294
Chapter 15	Using Semi-Continuous Variables: a Rates Example	295
	What Are Semi-Continuous Variables?	296
	Describing the Problem	296
	Representing the Problem	297
	Building a Model	297
	Solving the Problem	298
	Ending the Application	298
	Complete Program	298
Chapter 16	Using Piecewise Linear Functions in Optimization: a Transport Example	299

	Piecewise Linearity in ILOG CPLEX	300
	What Is a Piecewise Linear Function?.....	300
	Syntax of Piecewise Linear Functions.....	301
	Discontinuous Piecewise Linear Functions.....	301
	Isolated Points in Piecewise Linear Functions.....	303
	Using IloPiecewiseLinear.....	303
	Describing the Problem	304
	Variable Shipping Costs.....	304
	Model with Varying Costs.....	306
	Developing a Model	307
	Representing the Data.....	307
	Adding Constraints.....	307
	Checking Convexity and Concavity.....	308
	Adding an Objective.....	308
	Solving the Problem	309
	Displaying a Solution	309
	Ending the Application	309
	Complete Program: transport.cpp	309
Chapter 17	Logical Constraints in Optimization	311
	What Are Logical Constraints?	312
	What Can Be Extracted from a Model with Logical Constraints?	312
	Logical Constraints in the C++ API.....	312
	Logical Constraints in the Java API.....	313
	Logical Constraints in the .NET API.....	313
	Which Nonlinear Expressions Can Be Extracted?	314
	Logical Constraints for Counting	315
	Logical Constraints as Binary Variables	315
	How Are Logical Constraints Extracted?	315
Chapter 18	Using Indicator Constraints	317
	What Is an Indicator Constraint?	317

	Example: fixnet.c	318
	Indicator Constraints in the Interactive Optimizer	318
	What Are Indicator Variables?	319
	Restrictions on Indicator Constraints	319
	Best Practices with Indicator Constraints	319
Chapter 19	Using Logical Constraints: Food Manufacture 2	321
	Describing the Problem	322
	Representing the Data	322
	What Is Known?	323
	What Is Unknown?	323
	What Are the Constraints?	324
	What Is the Objective?	325
	Developing the Model	325
	Using Logical Constraints	327
	Solving the Problem	327
	Ending the Program	328
Chapter 20	Early Tardy Scheduling	329
	Describing the Problem	330
	Understanding the Data File	330
	Reading the Data	331
	Creating Variables	331
	Stating Precedence Constraints	332
	Stating Resource Constraints	332
	Representing the Piecewise Linear Cost Function	332
	Transforming the Problem	333
	Solving the Problem	334
Chapter 21	Using Column Generation: a Cutting Stock Example	335
	What Is Column Generation?	336
	Column-Wise Models in Concert Technology	336
	Describing the Problem	337

	Representing the Data	338
	Developing the Model: Building and Modifying	339
	Adding Extractable Objects: Both Ways	339
	Using a Template to Add Objects	339
	Using a Method to Add Objects	340
	Adding Columns to a Model	340
	Changing the Type of a Variable	341
	Cut Optimization Model	341
	Pattern Generator Model	342
	Changing the Objective Function	342
	Solving the Problem: Using More than One Algorithm	342
	Ending the Program	343
	Complete Program	344
Part V	Infeasibility and Unboundedness	345
Chapter 22	Preprocessing and Feasibility	347
Chapter 23	Managing Unboundedness	349
	What Is Unboundedness?	350
	Avoiding Unboundedness	350
	Diagnosing Unboundedness	351
Chapter 24	Diagnosing Infeasibility by Refining Conflicts	353
	What Is a Conflict?	353
	What a Conflict Is Not	354
	How to Invoke the Conflict Refiner	355
	How a Conflict Differs from an IIS	355
	Meet the Conflict Refiner in the Interactive Optimizer	356
	A Model for the Conflict Refiner	356
	Optimizing the Example	357
	Interpreting the Results and Detecting Conflict	357
	Displaying a Conflict	358

	Interpreting Conflict	359
	Deleting a Constraint	359
	Understanding a Conflict Report	360
	Summing Equality Constraints	360
	Changing a Bound	360
	Adding a Constraint	361
	Changing Bounds on Cost	362
	Relaxing a Constraint	362
	More about the Conflict Refiner	363
	Using the Conflict Refiner in an Application	365
	What Belongs in an Application to Refine Conflict	367
	Conflict Application vs Interactive Optimizer	367
	Preferences in the Conflict Refiner	368
	Groups in the Conflict Refiner	368
Chapter 25	Repairing Infeasibilities with FeasOpt	371
	What Is FeasOpt?	371
	Invoking FeasOpt	372
	Specifying Preferences	373
	Example: FeasOpt in Concert Technology	373
Part VI	Advanced Programming Techniques	379
Chapter 26	User-Cut and Lazy-Constraint Pools	381
	What Are Pools of User Cuts or Lazy Constraints?	382
	Adding User Cuts and Lazy Constraints	384
	Using Component Libraries	384
	Using the Interactive Optimizer	384
	Reading and Writing LP Files	384
	General Syntax	385
	Example	385
	Reading and Writing SAV Files	386

	Reading and Writing MPS Files	386
	Deleting User Cuts and Lazy Constraints	387
Chapter 27	Using Goals	389
	Branch & Cut with Goals	390
	Overview of Goals in Branch & Cut	390
	How Goals Are Implemented in Branch & Cut	391
	About the Method execute in Branch & Cut	391
	Special Goals in Branch & Cut	392
	Or Goal	392
	And Goal	392
	Fail Goal	392
	Local Cut Goal	393
	Null Goal	393
	Branch as CPLEX Goal	393
	Solution Goal	394
	Aggregating Goals	394
	Example: Goals in Branch & Cut	395
	The Goal Stack	397
	Memory Management and Goals	398
	Cuts and Goals	399
	Injecting Heuristic Solutions	401
	Controlling Goal-Defined Search	402
	Example: Using Node Evaluators in a Node Selection Strategy	404
	Search Limits	406
Chapter 28	Using Callbacks	407
	Diagnostic Callbacks	408
	Implementing Callbacks in ILOG CPLEX with Concert Technology	408
	Writing Callback Classes by Hand	409
	Writing Callbacks with Macros	410
	Callback Interface	412

	The Continuous Callback	412
	Example: Deriving the Simplex Callback ilolpex4.cpp	412
	Implementing Callbacks in the Callable Library	414
	Setting Callbacks	414
	Callbacks for Continuous and Discrete Problems	415
	Return Values for Callbacks.	415
	Interaction Between Callbacks and ILOG CPLEX Parallel Optimizers	416
	Example: Using Callbacks lpex4.c	416
	Control Callbacks for IloCplex	417
	Example: Controlling Cuts iloadmipex5.cpp.	418
Chapter 29	Goals and Callbacks: a Comparison	425
Chapter 30	Advanced Presolve Routines	429
	Introduction to Presolve	430
	A Proposed Example	431
	Restricting Presolve Reductions	432
	Adding Constraints to the First Solution.	432
	Primal and Dual Considerations in Presolve Reductions.	433
	Cuts and Presolve Reductions.	433
	Infeasibility or Unboundedness in Presolve Reductions	434
	Protected Variables in Presolve Reductions	434
	Manual Control of Presolve.	434
	Modifying a Problem	437
Chapter 31	Advanced MIP Control Interface	439
	Introduction to MIP Callbacks.	440
	Heuristic Callback	441
	Cut Callback	442
	Branch Selection Callback	443
	Incumbent Callback	444
	Node Selection Callback	445
	Solve Callback	445

Chapter 32	Parallel Optimizers	447
	Threads	448
	Example: Threads and Licensing	449
	Threads and Performance Considerations	449
	Nondeterminism	449
	Clock Settings and Time Measurement	450
	Using Parallel Optimizers in the Interactive Optimizer	450
	Using Parallel Optimizers in the ILOG CPLEX Component Libraries	451
	Parallel Barrier Optimizer	451
	Concurrent Optimizer	452
	Parallel MIP Optimizer	452
	Memory Considerations and the Parallel MIP Optimizer	454
	Output from the Parallel MIP Optimizer	454
Index		457

Meet ILOG CPLEX

ILOG CPLEX offers C, C++, Java, and .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 37

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 .NET applications. The library is provided in files `ilocplex91.lib`, `concert.lib` and `cplex91.jar` as well as `cplex91.dll` and `concert21.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:

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

where the relation \sim 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, \dots, c_n
constraint coefficients	a_{11}, \dots, a_{mn}
righthand side	b_1, \dots, b_m
upper bounds	u_1, \dots, u_n
lower bounds	l_1, \dots, l_n
variables or unknowns	x_1, \dots, 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 *Solving LPs: Simplex Optimizers* on page 161.

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. *Solving Problems with a Quadratic Objective (QP)* on page 217, 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. *Solving Problems with Quadratic Constraints (QCP)* on page 229, tells you more about the kinds of quadratically constrained problems that ILOG CPLEX solves, including the special case of second order cone programming (SOCP) problems.

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 *Solving Mixed Integer Programming Problems (MIP)* on page 245.

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. *Solving Network-Flow Problems* on page 207 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 read *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 37 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 .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, .NET, or others.

- ◆ 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, Infeasibility and Unboundedness

This part confronts unsatisfactory results of optimization, such as infeasibility of solutions or unboundedness of decision variables, and suggests ways of formulating or reformulating a model to eliminate or at least to minimize such obstacles.

◆ Part VI, 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

ILOG Concert Technology for C++ Users on page 41, 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.

ILOG Concert Technology for Java Users on page 69, 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.

ILOG Concert Technology for .NET Users on page 95, offers an example of this API.

ILOG CPLEX Callable Library on page 107, 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*

Developing CPLEX Applications on page 131, 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.

Managing Input and Output on page 141, 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.

Licensing an Application on page 153, tells you what you must consider when you want to license your ILOG CPLEX application for deployment.

Part III, *Continuous Optimization*

Solving LPs: Simplex Optimizers on page 161, 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.

Solving LPs: Barrier Optimizer on page 187, 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.

Solving Network-Flow Problems on page 207, shows how to use the ILOG CPLEX Network Optimizer on linear programming problems based on a network model.

Solving Problems with a Quadratic Objective (QP) on page 217, takes up programming problems in which the objective function may be quadratic. It, too, includes examples.

Solving Problems with Quadratic Constraints (QCP) on page 229, introduces problems where the constraints are not strictly linear but may also include convex quadratic constraints and shows how to use the barrier optimizer to solve them.

Part IV, *Discrete Optimization*

Solving Mixed Integer Programming Problems (MIP) on page 245, shows you how to handle MIPs. It particularly emphasizes performance tuning and offers a series of examples.

Using Special Ordered Sets (SOS) on page 291, sketches how to declare and use special ordered sets in formulating your model.

Using Semi-Continuous Variables: a Rates Example on page 295, demonstrates how to use semi-continuous variables in a rate-setting problem.

Using Piecewise Linear Functions in Optimization: a Transport Example on page 299, applies piecewise linear functions to model a transportation problem.

Logical Constraints in Optimization on page 311, introduces logical constraints as they are implemented in Concert Technology and in the Callable Library.

Using Indicator Constraints on page 317, explains a technique for expressing relations among constraints in a model by means of binary variables that turn on or off enforcement of a given constraint.

Using Logical Constraints: Food Manufacture 2 on page 321, follows up that introduction to logical constraints with an example borrowed from a well known textbook about modeling.

Early Tardy Scheduling on page 329, demonstrates logical constraints, piecewise linear functions in optimization, and aggressive MIP emphasis in a production planning example that includes penalties for earliness and tardiness.

Using Column Generation: a Cutting Stock Example on page 335, shows how to formulate a model by generating columns one by one. It uses a cutting stock example to illustrate the technique.

Part V, Infeasibility and Unboundedness

Preprocessing and Feasibility on page 347 introduces you to the effects of preprocessing on feasibility and infeasibility.

Managing Unboundedness on page 349 explains what a report of unbounded means, suggests ways to avoid an unbounded outcome, and outlines means to diagnose the cause of unboundedness in your model.

Diagnosing Infeasibility by Refining Conflicts on page 353, describes the conflict refiner, a feature of ILOG CPLEX that helps you identify contradictory constraints and bounds within your model.

Repairing Infeasibilities with FeasOpt on page 371, documents a feature of ILOG CPLEX that may enable you to repair detected infeasibilities in your model.

Part VI, Advanced Programming Techniques

User-Cut and Lazy-Constraint Pools on page 381, formerly available only through Customer Support, is now part of the standard documentation. It explains in greater detail how to manage your own pools of cuts and lazy constraints.

Using Goals on page 389, shows how to use goals to control a MIP search.

Using Callbacks on page 407 shows how to use callbacks to control a MIP search.

Goals and Callbacks: a Comparison on page 425, compares the two different approaches.

Advanced Presolve Routines on page 429, formerly available only through Technical Support, is now part of the standard documentation. It documents advanced aspects of presolve and aggregation more fully.

Advanced MIP Control Interface on page 439, 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.

Parallel Optimizers on page 447, explains how to exploit parallel optimizers in case your hardware supports parallel execution.

The *Index* on page 457 completes this manual.

Examples Online

For the examples explained in the manual, you will find the complete code for the solution in the `examples` subdirectory of the standard distribution of ILOG CPLEX, so that you can see exactly how ILOG CPLEX fits into your own applications. Table 1 lists the examples in this manual and indicates where to find them.

Table 1 Examples

Example	Source File	In This Manual
dietary optimization: building a model by rows (constraints) or by columns (variables), solving with <code>IloCplex</code> in C++	<code>ilodiet.cpp</code>	<i>Example: Optimizing the Diet Problem in C++</i> on page 63
dietary optimization: building a model by rows (constraints) or by columns (variables), solving with <code>IloCplex</code> in Java	<code>Diet.java</code>	<i>Example: Optimizing the Diet Problem in Java</i> on page 91
dietary optimization: building a model by rows (constraints) or by columns (variables), solving with <code>Cplex</code> in C#.NET	<code>Diet.cs</code>	<i>Example: Optimizing the Diet Problem in C#.NET</i> on page 105
dietary optimization: building a model by rows (constraints) or by columns (variables), solving with the Callable Library	<code>diet.c</code>	<i>Example: Optimizing the Diet Problem in the Callable Library</i> on page 123
linear programming: starting from an advanced basis	<code>ilolpex6.cpp</code> <code>lpex6.c</code>	<i>Example <code>ilolpex6.cpp</code></i> on page 184 <i>Example <code>lpex6.c</code></i> on page 184
network optimization: using the Callable Library	<code>netex1.c</code>	<i>Example: Using the Network Optimizer with the Callable Library <code>netex1.c</code></i> on page 213

Table 1 Examples

Example	Source File	In This Manual
network optimization: relaxing a network flow to an LP	netex2.c	<i>Example: Network to LP Transformation netex2.c on page 216</i>
quadratic programming: maximizing a QP	iloqpex1.cpp QPex1.java qpex1.c	<i>Example: iloqpex1.cpp on page 226</i> <i>Example: QPex1.java on page 227</i> <i>Example: qpex1.c on page 227</i>
quadratic programming: reading a QP from a formatted file	qpex2.c	<i>Example: Reading a QP from a File qpex2.c on page 228</i>
quadratically constrained programming: QCP	qcpex1.c iloqcpx1.cpp QCPex1.java	<i>Examples: QCP on page 241</i>
mixed integer programming: optimizing a basic MIP	ilomipex1.cpp mipex1.c	<i>Examples: Optimizing a Basic MIP Problem on page 288</i>
mixed integer programming: reading a MIP from a formatted file	ilomipex2.cpp mipex2.c	<i>Example: Reading a MIP Problem from a File on page 289</i>
mixed integer programming: using special ordered sets (SOS) and priority orders	ilomipex3.cpp mipex3.c	<i>Examples: Using SOS and Priority on page 293</i>
cutting stock: using column generation	cutstock.cpp	<i>What Is Column Generation? on page 336</i>
transport: piecewise-linear optimization	transport.cpp	<i>Complete Program: transport.cpp on page 309</i>
food manufacturing 2: using logical constraints	foodmanufac.cpp	<i>Using Logical Constraints: Food Manufacture 2 on page 321</i>
early tardy scheduling	etsp.cpp	<i>Early Tardy Scheduling on page 329</i>
input and output: using the message handler	lpex5.c	<i>Example: Callable Library Message Channels on page 150</i>
using query routines	lpex7.c	<i>Example: Using Query Routines lpex7.c on page 127</i>
using callbacks	ilolpex4.cpp lpex4.c iloadmipex5.cpp	<i>Example: Deriving the Simplex Callback ilolpex4.cpp on page 412</i> <i>Example: Using Callbacks lpex4.c on page 416</i> <i>Example: Controlling Cuts iloadmipex5.cpp on page 418</i>

Notation in This Manual

Like the reference manuals, 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 .NET classes and interfaces are the same as the corresponding entity in Java, except the name is **not** prefixed by `Ilo`. Names of .NET methods are the same as Java methods, except the .NET name is capitalized (that is, uppercase) to conform to Microsoft naming conventions.
- ◆ Where use of a specific language (C++, Java, C, C#, 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: ***AA*^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. On UNIX platforms, they can be found in `yourCplexHome/doc`. On Windows platforms, the online documentation can be found in the ILOG Optimization suite, for example, in `Start > Programs > ILOG > Optim` or in `C:\ILOG\Optim`.

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 .NET applications as well as the ILOG CPLEX Callable Library.

ILOG CPLEX Getting Started 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), including the special case of second order cone programming (SOCP) problems, 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 Reference Manual:** This manual supplies detailed definitions of the routines, macros, and functions in the ILOG CPLEX Callable Library C application programming interface (API). It is available online as HTML and as Microsoft compiled HTML help (.chm). The routines are organized into groups, such as `optim.cplex.callable.optimizers`, `optim.callable.debug`, or `optim.cplex.callable.callbacks`, to help you locate routines by their purpose.

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 .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 access 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 access solution status codes by number in the *Overview of the API* through the link *Interpreting Solution Status Codes*.
- ◆ **ILOG CPLEX C++ API Reference Manual:** This manual supplies detailed definitions of the classes, macros, and functions in the ILOG CPLEX C++ application programming interface (API). It is available online as HTML and as Microsoft compiled HTML help (.chm).
- ◆ **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 .NET Reference Manual:** This manual documents the .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 and Updates

The electronic mailing list 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 *Subscribe to 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 product 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.

Chvatal, Vasek, *Linear Programming*, New York: W.H. Freeman and Company, 1983. This standard textbook for undergraduate students introduces both theory and practice of linear programming.

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 .NET. This part contains:

- ◆ *ILOG Concert Technology for C++ Users* on page 41
- ◆ *ILOG Concert Technology for Java Users* on page 69
- ◆ *ILOG Concert Technology for .NET Users* on page 95
- ◆ *ILOG CPLEX Callable Library* on page 107

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:

- ◆ *Architecture of a CPLEX C++ Application* on page 42, including information about licensing, compiling, and linking your programs
- ◆ *Creating a C++ Application with Concert Technology* on page 43
- ◆ *Modeling an Optimization Problem with Concert Technology* on page 43
- ◆ *Solving the Model* on page 48
- ◆ *Accessing Solution Information* on page 55
- ◆ *Modifying a Model* on page 59
- ◆ *Handling Errors* on page 61
- ◆ *Example: Optimizing the Diet Problem in C++* on page 63

Architecture of a CPLEX C++ Application

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. **Solving objects** in an instance of `IloCplex` are used to solve models created with the modeling objects. An instance of `IloCplex` reads a model and extracts its data to the appropriate representation for the ILOG CPLEX optimizers. Then the `IloCplex` object is ready to solve the model it extracted. After it solves a model, it can be queried for solution information.

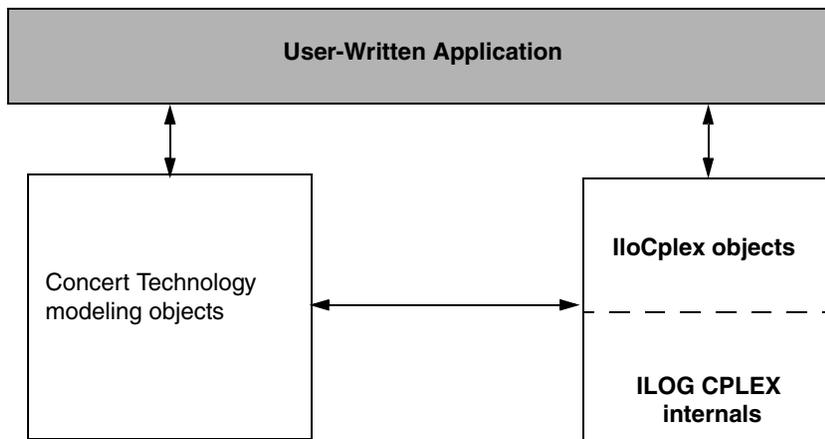


Figure 1.1 A View of Concert Technology for C++ Users

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. For details about contacting ILOG support, click "Customer Support" at the bottom of the first page of ILOG CPLEX online documentation.

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 a C++ 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 the Model* on page 48 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 55, shows you how to access and interpret results from the optimization after solving the model.
- ◆ After analyzing the results, you may want to make changes to the model and study their effect. *Modifying a Model* on page 59 explains how to make changes and how ILOG CPLEX deals with them.
- ◆ *Handling Errors* on page 61, discusses the error handling and debugging support provided by Concert Technology and ILOG CPLEX.
- ◆ *Example: Optimizing the Diet Problem in C++* on page 63 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 VI, *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. They are summarized in Table 1.1 on page 49.

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 *Numeric Difficulties* on page 174.

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:

- *Solving LPs: Simplex Optimizers* on page 161 and *Solving LPs: Barrier Optimizer* on page 187 both discuss linear expressions.
- *Solving Problems with a Quadratic Objective (QP)* on page 217 discusses quadratic expressions in an objective function.
- *Solving Problems with Quadratic Constraints (QCP)* on page 229 discusses quadratic expressions in quadratically constrained programming problems (QCPs), including the special case of second order cone programming (SOCP) problems.
- *Using Piecewise Linear Functions in Optimization: a Transport Example* on page 299 introduces piecewise-linear expressions through a transportation example.
- *Logical Constraints in Optimization* on page 311 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 \leq expression \leq 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, such as `IloNumSet`, provided by Concert Technology.

For example, objects of class `IloNumArray` can be used to store numeric 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 63) 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 the Model

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 *ILOG CPLEX Reference Manual*. Some models might also be solved by using other algorithms, such as the class `IloCP` for constraint programming, or by using a hybrid algorithm consisting of both ILOG CP or ILOG Solver 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. Constraints and ranges may also include piecewise linear terms. (Other sections of this manual, not specific to C++, show you how to use quadratic expressions: *Solving Problems with a Quadratic Objective (QP)* on page 217 and *Solving Problems with Quadratic Constraints (QCP)* on page 229. Likewise, *Using Piecewise Linear Functions in Optimization: a Transport Example* on page 299 shows you how to apply piecewise linear terms in a C++ application.)

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

- ◆ *Extracting a Model* on page 49
- ◆ *Invoking a Solver* on page 50
- ◆ *Choosing an Optimizer* on page 51
- ◆ *Controlling the Optimizers* on page 53

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

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

For an explanation of quadratic constraints, see *Solving Problems with Quadratic Constraints (QCP)* on page 229. For more information about quadratic objective functions, see *Solving Problems with a Quadratic Objective (QP)* on page 217. For examples of piecewise linear constraints, see *Using Piecewise Linear Functions in Optimization: a Transport Example* on page 299. For more about logical constraints, see *Logical Constraints in Optimization* on page 311. For a description of special ordered sets, see *Using Special Ordered Sets (SOS)* on page 291.

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 efficient 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 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.

Invoking a Solver

After the model is extracted to the `cplex` object, you are ready to solve it 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. *Solving LPs: Simplex Optimizers* on page 161 and *Solving LPs: Barrier Optimizer* on page 187 discuss the algorithms available for solving LPs. Similarly, *Solving Problems with a Quadratic Objective (QP)* on page 217, discusses the algorithms available for solving QPs. *Solving Problems with Quadratic Constraints (QCP)* on page 229 re-introduces the barrier optimizer in the context of quadratically constrained programming problems (QCPs). *Using Piecewise Linear Functions in Optimization: a Transport Example* on page 299 introduces piecewise-linear functions through a transportation example. *Logical Constraints in Optimization* on page 311 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. *Solving Mixed Integer Programming Problems (MIP)* on page 245 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 `IloCplex::Algorithm`.

As a nested enumeration, 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`.

The choice `Sifting` is **not** available for QP models. Only the `Barrier` option is available for QCP models. Table 1.3 on page 52 summarizes these options.

Table 1.2 *Optimizer Options in IloCplex::Algorithm*

<code>IloCplex::AutoAlg</code>	let CPLEX decide which algorithm to use
<code>IloCplex::Primal</code>	use the primal simplex algorithm
<code>IloCplex::Dual</code>	use the dual simplex algorithm
<code>IloCplex::Network</code>	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
<code>IloCplex::Barrier</code>	use the barrier algorithm. The type of crossover performed after the barrier algorithm is determined by parameter <code>IloCplex::BarCrossAlg</code> .
<code>IloCplex::Sifting</code>	use the sifting algorithm
<code>IloCplex::Concurrent</code>	use multiple algorithms concurrently on a multiprocessor system

Table 1.3 *Algorithm Available at Root by Problem Type*

Value	Algorithm Type	LP? MILP?	QP? MIQP?	QCP? MIQCP?
0	<code>IloCplex::AutoAlg</code>	yes	yes	yes
1	<code>IloCplex::Primal</code>	yes	yes	not available
2	<code>IloCplex::Dual</code>	yes	yes	not available
3	<code>IloCplex::Network</code>	yes	yes	not available
4	<code>IloCplex::Barrier</code>	yes	yes	yes
5	<code>IloCplex::Sifting</code>	yes	not available	not available
6	<code>IloCplex::Concurrent</code>	yes	yes	not available

If the extracted model requires the solution of more than one continuous relaxation, the algorithm for solving the first one at the root is controlled by the `RootAlg` parameter. The algorithm at all other nodes except the root is controlled by the `NodeAlg` parameter:

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

Table 1.4 on page 53 summarizes the options available at nodes.

Table 1.4 *Algorithm Types for NodeAlg*

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

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 1.5 summarizes those parameters and their enumeration types.

Table 1.5 *Integer Parameters Tied to Nested Enumerations*

This Enumeration:	Is Used for This Parameter:
<code>IloCplex::Algorithm</code>	<code>IloCplex::RootAlg</code>
<code>IloCplex::Algorithm</code>	<code>IloCplex::NodeAlg</code>
<code>IloCplex::MIPEmphasisType</code>	<code>IloCplex::MIPEmphasis</code>
<code>IloCplex::VariableSelect</code>	<code>IloCplex::VarSel</code>
<code>IloCplex::NodeSelect</code>	<code>IloCplex::NodeSel</code>
<code>IloCplex::PrimalPricing</code>	<code>IloCplex::PPriInd</code>
<code>IloCplex::DualPricing</code>	<code>IloCplex::DPriInd</code>
<code>IloCplex::BranchDirection</code>	<code>IloCplex::BrDir</code>

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 numeric parameter `IloCplex::EpOpt` controlling the optimality tolerance for the simplex algorithms can be set to 0.0001 by calling

```
cpplex.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 `setDefault` *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 265. 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 389, and in *Using Callbacks* on page 407. 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 425 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 55
- ◆ *Querying Solution Data* on page 56
- ◆ *Accessing Basis Information* on page 57
- ◆ *Performing Sensitivity Analysis* on page 57
- ◆ *Analyzing Infeasible Problems* on page 57
- ◆ *Solution Quality* on page 58

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 `IloAlgorithm::Status`. The fully qualified names of those symbols have the `IloAlgorithm` prefix. Table 1.6 shows what each return status means for the extracted model.

Table 1.6 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 <code>IloCplex</code> 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`.

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 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 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. In one approach, known as FeasOpt, ILOG CPLEX accepts an infeasible model and selectively relaxes bounds and constraints to find a minimal set of changes that would make the model feasible. It then reports these suggested changes and the solution they would produce for you to decide whether to apply them in your model. For more about this approach, see *Repairing Infeasibility: FeasOpt* on page 183.

In another approach, ILOG CPLEX can detect a conflict among the constraints and bounds of an infeasible model and refine the conflict to report to you a minimal conflict to repair yourself. For more about this approach, see *Diagnosing Infeasibility by Refining Conflicts* on page 353.

For more about these and other ways of overcoming infeasibility, see *Diagnosing LP Infeasibility* on page 179.

Solution Quality

The ILOG CPLEX optimizer uses finite precision arithmetic to compute solutions. To compensate for numeric 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 analyze a scenario 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. To do so, call the modification methods of the individual modeling objects.

When an extracted model is modified, the modification is tracked in the `cplex` object 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? If you call its `end` method, 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: call the method `removeFromAll`. This method may be helpful to temporarily remove a variable from your model while keeping the option to add it back later.

It is important to understand the difference between calling `end` and calling `model.remove(obj)` for an object `obj`. In the case of a call to `remove`, `obj` is not necessarily removed from the problem ILOG CPLEX maintains. Whether or not anything appears to happen depends on whether the removed object is referenced by yet another extracted modeling object. For example, when you add 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 you remove that modeling object (for example, that ranged constraint), those variables are not implicitly removed because they may be referenced by other elements (such as the objective function or a basis, for example). For that reason, variables can be explicitly removed from a model only by a call to its `end` member function.

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 `end` or `remove` operations. 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 specify a type only 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 convention is not too restrictive, since you can remove the conversion from a model and add a new one. To remove a conversion from a model, use the method `IloExtractable::end`. To add a new one, use the method `IloModel::add`. For a sample of code using these methods in this procedure, see the documentation of the class `IloConversion` in the *ILOG CPLEX C++ Reference Manual*.

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. After they are recognized and fixed there is usually no danger of corrupting an application. In a production application, 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 application 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 application. 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. *Problem Representation* on page 63 describes the problem.

The example `ilodiet.cpp` illustrates these procedures:

- ◆ *Creating a Model Row by Row* on page 63;
- ◆ *Creating a Model Column by Column* on page 64.

To continue this example, *Application Description* on page 65 outlines the structure of the application. The following sections explain more about data structures useful in this application.

- ◆ *Creating Multi-Dimensional Arrays with `IloArray`* on page 65;
- ◆ *Using Arrays for Input/Output* on page 65;
- ◆ *Solving the Model with `IloCplex`* on page 67.

Notes about the application are available in *Complete Program* on page 68.

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 nutrition 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 in pseudo-code like this:

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

where i represents the number of the nutrient under consideration, $\text{nutrMin}[i]$ and $\text{nutrMax}[i]$ the minimum and maximum amount of nutrient i and $\text{nutrPer}[i][j]$ the amount of nutrient i in food j .

Finally, you specify your objective function in pseudo-code like this:

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

The loop in the example combines those two ideas and looks like this:

```
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();
}
```

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

Creating a Model Column by Column

You start with the nutrition 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 \dots 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 the variable in the objective function and constraints. That is, you create the following column in pseudo code, like this:

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

where the notation `+` and `sum` indicate in pseudo code 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 the example `ilodiet.cpp`.

```
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();
}
```

Application Description

In `ilodiet.cpp`, 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.*

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, `IloNumArray2`.

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 $\sum_j (\text{Buy}[j] * \text{foodCost}[j])$ which is then passed to the function `IloMinimize`. That function creates and returns the actual `IloObjective` object, which is added to the model with the call `mod.add`.

The following loop creates the constraints of the problem one by one and adds them to the model. First the expression $\sum_j (\text{Buy}[j] * \text{nutrPer}[i][j])$ 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` is available online in the standard distribution at `yourCPLEXinstallation/examples/src`.

Notes:

- ◆ *All the definitions needed for an ILOG CPLEX Concert Technology application are imported by including the file `<ilcplex/ilcplex.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.*

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.

- ◆ *Architecture of a CPLEX Java Application* on page 70
- ◆ *Modeling an Optimization Problem with Concert Technology* on page 72
- ◆ *Building the Model* on page 77
- ◆ *Solving the Model* on page 79
- ◆ *Accessing Solution Information* on page 80
- ◆ *Choosing an Optimizer* on page 81
- ◆ *Controlling ILOG CPLEX Optimizers* on page 84
- ◆ *More Solution Information* on page 86
- ◆ *Advanced Modeling with IloLPMatrix* on page 89
- ◆ *Modeling by Column* on page 90
- ◆ *Example: Optimizing the Diet Problem in Java* on page 91

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 accesses the variable only 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 internals. The Java interfaces, represented by the dashed outline, do not actually consume memory. The ILOG CPLEX internals include 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.

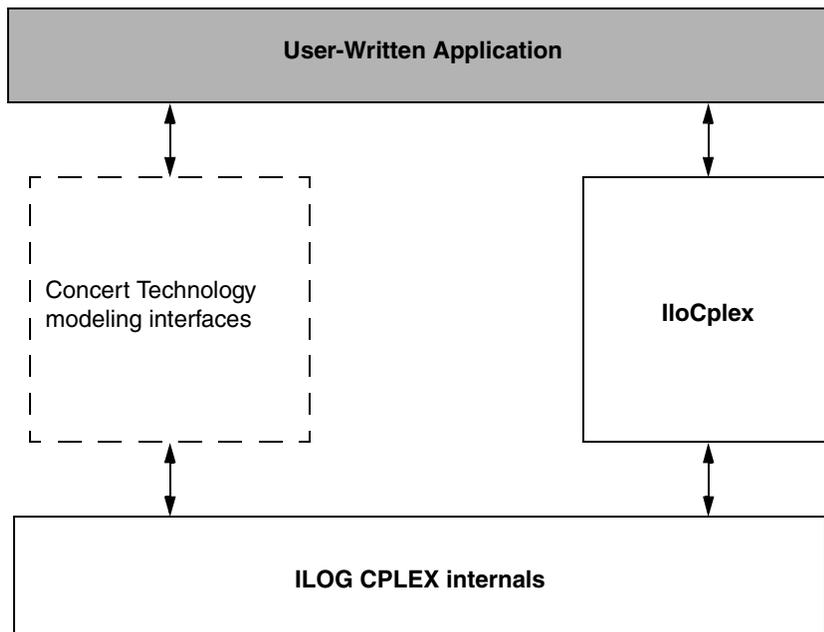


Figure 2.1 A View of Concert Technology for Java Users

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. For details about contacting ILOG support, click "Customer Support" at the bottom of the first page of ILOG CPLEX online documentation.

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 file `readme.html` for details.

Creating a Java Application with Concert Technology

This chapter covers the steps most Java 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 72 offers an introduction to creating a model. *Building the Model* on page 77 goes into more detail.

When the model is ready to be solved, hand it over to ILOG CPLEX for solving. *Solving the Model* on page 79 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 80 shows you how to access and interpret results from the optimization after solving the model.

After analyzing the results, you may want to make changes to the model and study their effect. *Modifying the Model* on page 92 explains how to make changes and how ILOG CPLEX deals with them in the context of the diet problem.

Example: Optimizing the Diet Problem in Java on page 91 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 *Advanced Programming Techniques* on page 379.

Modeling an Optimization Problem with Concert Technology

An optimization problem is represented by a set of interconnected modeling objects in an instance of `IloCplex` or `IloCplexModeler`. 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 `IloMPSModeler` or `IloCPModeler`.

Notes:

The class `IloCplex` extends `IloCplexModeler`. All the modeling methods in `IloCplex` derive from `IloCplexModeler`. `IloCplex` implements the solving methods.

The class `IloCplexModeler`, which implements `IloMPSModeler`, makes it possible for a user to build models in a Java application as pure Java objects, without using the class `IloCplex`.

In particular, a model built with `IloCplexModeler` using no instance of `IloCplex` does not require loading of the `CPLEX.dll` nor any shared library.

Furthermore, `IloCplexModeler` is serializable. For example, a user may develop a pure Java application that builds a model with `IloCplexModeler` and sends the model and modeling objects off to an optimization server that uses `IloCplex`.

The example `CplexServer.java` shows you how to write an optimization server that accepts pure Java model taking advantage of the class `IloCplexModeler` in a native J2EE client application.

This discussion concentrates on `IloModeler` and `IloMPSModeler` because the classes `IloCplex` and `IloCplexModeler` implement these interfaces and thus inherit 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 `IloMPSModeler` (and thus its parent interface `IloModeler`) all methods from `IloMPSModeler` 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:
$$\text{lowerbound} \leq \text{expression} \leq \text{upperbound}$$

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 `IloMPSModeler` extension.

The `IloMPSModeler` 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 or Interface
variable	<code>IloNumVar</code> and its extensions <code>IloIntVar</code> and <code>IloSemiContVar</code>
range constraint	<code>IloRange</code> with (piecewise) linear or quadratic expressions
other relational constraint	<code>IloConstraint</code> of the form <i>expr1 relation expr2</i> , where both expressions are linear or quadratic and may optionally contain piecewise linear terms.
LP matrix	<code>IloLPMatrix</code>
linear or quadratic objective	<code>IloObjective</code> with (piecewise) linear or quadratic expressions
variable type-conversion	<code>IloConversion</code>
special ordered set	<code>IloSOS1</code> or <code>IloSOS2</code>
logical constraints	<code>IloOr</code> , <code>IloAnd</code> , and methods such as <code>not</code>

For an explanation of quadratic constraints, see *Solving Problems with Quadratic Constraints (QCP)* on page 229. For more information about quadratic objective functions, see *Solving Problems with a Quadratic Objective (QP)* on page 217. For examples of piecewise linear constraints, see *Using Piecewise Linear Functions in Optimization: a Transport Example* on page 299. For a description of special ordered sets, see *Using Special Ordered Sets (SOS)* on page 291. For more about logical constraints, see *Logical Constraints in Optimization* on page 311.

Using 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 `IloMIPModeler` 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

$$x_1 + 2 \cdot x_2$$

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 *Numeric Difficulties* on page 174.

Ranged Constraints

Ranged constraints are constraints of the form: $lb \leq \text{expression} \leq ub$ 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 $=, \leq,$ or \geq . The following table shows how to choose `lb` and `ub` for modeling these relations:

relation	lb	ub	method
=	rhs	rhs	eq
\leq	<code>-Double.MAX_VALUE</code>	rhs	le
\geq	rhs	<code>Double.MAX_VALUE</code>	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 righthand side (RHS). For example, the constraint $\text{expr} \leq 1.0$ 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 *Using IloModeler* on page 74, 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 `addConstructorName` 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 `addConstrucorName` 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:

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

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

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

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

$$\text{foodMin}[j] \leq \text{Buy}[j] \leq \text{foodMax}[j]$$

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.
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 <code>IloCplex</code> 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 (SOS);
 - 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 *Solving LPs: Simplex Optimizers* on page 161, *Solving LPs: Barrier Optimizer* on page 187, and *Solving Network-Flow Problems* on page 207 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 *Solving Problems with a Quadratic Objective (QP)* on page 217.

- ◆ 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 *Solving Problems with Quadratic Constraints (QCP)* on page 229, where the special case of second order cone programming (SOCP) problems is also discussed.

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 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. If it includes a positive semidefinite quadratic term in the objective, it is called an MIQP. If it includes a constraint that contains a positive semidefinite quadratic term, it is called an MIQCP. MIPs are solved using branch & cut search, explained in more detail in *Solving Mixed Integer Programming Problems (MIP)* on page 245.

Solving a Single Continuous Model

To choose the optimizer to solve a single continuous 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	<code>IloCplex.Algorithm.Auto</code>	yes	yes	yes
1	<code>IloCplex.Algorithm.Primal</code>	yes	yes	not available
2	<code>IloCplex.Algorithm.Dual</code>	yes	yes	not available
3	<code>IloCplex.Algorithm.Network</code>	yes	yes	not available
4	<code>IloCplex.Algorithm.Barrier</code>	yes	yes	yes
5	<code>IloCplex.Algorithm.Sifting</code>	yes	not available	not available
6	<code>IloCplex.Algorithm.Concurrent</code>	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`. In contrast, any invalid setting, such as a value other than those of the enumeration, 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 available 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

continuous 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	<code>IloCplex.Algorithm.Auto</code>	yes	yes	yes
1	<code>IloCplex.Algorithm.Primal</code>	yes	yes	not available
2	<code>IloCplex.Algorithm.Dual</code>	yes	yes	not available
3	<code>IloCplex.Algorithm.Network</code>	yes	not available	not available
4	<code>IloCplex.Algorithm.Barrier</code>	yes	yes	yes
5	<code>IloCplex.Algorithm.Sifting</code>	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 <code>IloCplex.PrimalPricing</code>
0	<code>IloCplex.PrimalPricing.Auto</code>
1	<code>IloCplex.PrimalPricing.Devex</code>
2	<code>IloCplex.PrimalPricing.Steep</code>
3	<code>IloCplex.PrimalPricing.SteepQStart</code>
4	<code>IloCplex.PrimalPricing.Full</code>

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:
<code>IloCplex.Algorithm</code>	<code>IloCplex.IntParam.RootAlg</code> <code>IloCplex.IntParam.NodeAlg</code>
<code>IloCplex.MIPEmphasis</code>	<code>IloCplex.IntParam.MIPEmphasis</code>
<code>IloCplex.VariableSelect</code>	<code>IloCplex.IntParam.VarSel</code>
<code>IloCplex.NodeSelect</code>	<code>IloCplex.IntParam.NodeSel</code>
<code>IloCplex.DualPricing</code>	<code>IloCplex.IntParam.DPriInd</code>
<code>IloCplex.PrimalPricing</code>	<code>IloCplex.IntParam.PPriInd</code>

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 CPLEX is solving a MIP, another important way for you 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 389. Control callbacks are discussed in *Using Callbacks* on page 407. (Java goals and callbacks are similar to the C++ goals and callbacks.) 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 425 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 86
- ◆ *Dual Solution Information* on page 87
- ◆ *Basis Information* on page 87
- ◆ *Writing Solution Files* on page 86
- ◆ *Infeasible Solution Information* on page 88
- ◆ *Solution Quality* on page 88

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 187.

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 through two different approaches: the conflict refiner and `FeasOpt`.

One approach, invoked by the method `IloCplex.refineConflict`, computes a minimal set of conflicting constraints and bounds and reports them to you for you to take action to remove the conflict from your infeasible model. For more about this approach, see *Diagnosing Infeasibility by Refining Conflicts* on page 353.

Another approach to consider is the method `IloCplex.feasOpt` to explore whether there are modifications you can make that would render your model feasible. *Repairing Infeasibility: FeasOpt* on page 183 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 numeric 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 `getQuality`, like this:

```
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 rows*. 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 `IloMPSModeler` 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 `IloMPSModeler.column` methods. Several `IloColumn` objects can be linked together (with the `IloCplex.and` 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.

Example: 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 63
- ◆ *Example: Optimizing the Diet Problem in C#.NET* on page 105
- ◆ *Example: Optimizing the Diet Problem in the Callable Library* on page 123

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.

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 `IloMIPModeler` 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 memory that ILOG CPLEX uses.

The entire source code listing for the example is available as `Diet.java` in the standard distribution at `yourCPLEXinstallation/examples/src`.

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, you can use `IloModel.remove` to remove a modeling object from a model, if you have previously added that object.

When you add 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 you remove a modeling object, no variables are implicitly removed from the model. Instead, variables can only be explicitly removed from a model by calling `IloMPSModeler.delete`. (The interface `IloMPSModeler` derives from the class `IloModel`, among others. It is implemented by the class `IloCplex`.) This call 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, you can change variable bounds by using the methods `IloNumVar.setLB` and `IloNumVar.setUB`. Similarly, you can change the bounds of ranged constraints by 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 `IloMPSModeler`. These methods are for manipulating the linear expressions in ranged constraints and objective functions used with `IloCplex`. The methods `IloMPSModeler.setLinearCoef`, `IloMPSModeler.setLinearCoefs`, and `IloMPSModeler.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 be used only in at most one conversion object, or the model will no longer be unambiguously defined. This convention does not imply that the type of a variable can be changed only once and never again after that. Instead, you can remove the conversion object and add a new one to implement consecutive variable type changes. To remove the conversion object, use the method `IloModel.remove`.

ILOG Concert Technology for .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 96 contains the problem description.
- ◆ *Model* on page 98 shows how to represent the problem.
- ◆ *Solve* on page 102 demonstrates how to solve the problem and display the solution.
- ◆ *Good Programming Practices* on page 103 adds other features of the working application.
- ◆ *Example: Optimizing the Diet Problem in C#.NET* on page 105 tells you where to find the complete application and problem data.

The .NET API can be used from any programming 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\src`. 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.

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 not exceed what is available.

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 99, 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\x86_.net2003_7.1\format\Diet.csproj`

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

`yourCPLEXhome\examples\tutorials\DietLesson.cs`

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 108, including information about licensing and about compiling and linking your programs
- ◆ *Using the Callable Library in an Application* on page 109
- ◆ *ILOG CPLEX Programming Practices* on page 112
- ◆ *Managing Parameters from the Callable Library* on page 121
- ◆ *Example: Optimizing the Diet Problem in the Callable Library* on page 123
- ◆ *Using Surplus Arguments for Array Allocations* on page 126
- ◆ *Example: Using Query Routines lpex7.c* on page 127

Architecture of the ILOG CPLEX Callable Library

ILOG CPLEX includes a callable C library that makes it possible 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.

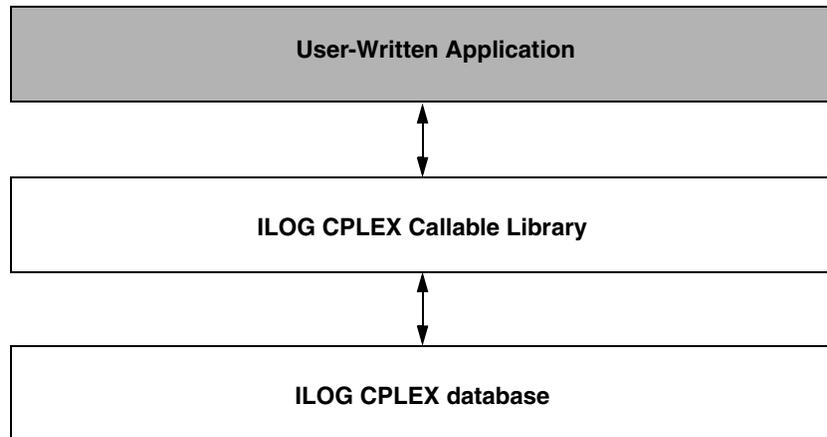


Figure 4.1 A View of the ILOG CPLEX Callable Library

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. For details about contacting ILOG support, click "Customer Support" at the bottom of the first page of ILOG CPLEX online documentation.

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, quadratic, or mixed integer programming problems (LPs, QPs, or MIPs)—efficiently by reusing the current feasible solution (basis or incumbent) 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 it 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:
 - `CPXaddcols`
 - `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 142 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 *Numeric Difficulties* on page 174.

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 162 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. Use the routines whose names begin with `CPXchg` to modify existing objects in the model, or use the routines `CPXaddcols`, `CPXaddqconstr`, `CPXaddrows`, `CPXnewcols`, and `CPXnewrows` to add new constraints and new variables to the model.

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 routine `CPXchgbds`. 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 121 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
<code>CPXENVptr</code> <code>CPXCENVptr</code>	ILOG CPLEX environment	<code>CPXENVptr env;</code>	<code>CPXopenCPLEX</code>
<code>CPXLPptr</code> <code>CPXCLPptr</code>	problem object	<code>CPXLPptr lp;</code>	<code>CPXcreateprob</code>
<code>CPXNETptr</code> <code>CPXCNETptr</code>	problem object	<code>CPXNETptr net;</code>	<code>CPXNETcreateprob</code>
<code>CPXCHANNELptr</code>	message channel	<code>CPXCHANNELptr channel;</code>	<code>CPXgetchannels</code> <code>CPXaddchannel</code>

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 111, 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`.

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 its 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 4.2 shows you the names and purpose of these

Table 4.2 *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

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, `plex.h`. *Managing Parameters from the Callable Library* on page 121 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 255 characters. Routines from the Callable Library truncate strings at 255 characters in output *text* files (such as MPS or LP 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. 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` (int) to the value `CPX_ON` to activate additional checking of array arguments for `CPXcopyData`, `CPXreadData`, and `CPXchgData` routines (where *Data* varies). The additional checks include:

- ◆ invalid *sense/ctype/sostype* values
- ◆ indices out of range, for example, `rowind` \geq `numrows`
- ◆ duplicate entries
- ◆ `matbeg` or `sosbeg` array with decreasing values
- ◆ NANS in double arrays
- ◆ NULLs in name arrays

This additional checking may entail overhead (time and memory). 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 407 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 416 and *Example: Callable Library Message Channels* on page 150.

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 149 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 product, 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-*insensitive* 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, `plex.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 `plex.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. *Problem Representation* on page 123 describes the problem to be solved.

The example `diet.c` illustrates these points:

- ◆ *Creating a Model Row by Row* on page 124;
- ◆ *Creating a Model Column by Column* on page 124;
- ◆ *Solving the Model with CPXlpopt* on page 125.

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, 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 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

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

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

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

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:

$$\text{nutrmin}[i] \leq \dots \leq \text{nutrmax}[i]$$

where \dots 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:

$$\text{cost}(\text{foodCost}[j]) \text{ "+" "sum_i" } (\text{rng}[i] (\text{nutrper}[i][j]))$$

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.

Here's another way to visualize a column, such as column *j* in this example:

```
foodCost [j]
nutrper [0] [j]
nutrper [1] [j]
...
nutrper [m-1] [j]
```

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.

The complete program, `diet.c`, appears online in the standard distribution at `yourCPLEXinstallation/examples/src`.

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 (zero) 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 Ipex7.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 126. 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 online in the standard distribution at *yourCPLEXinstallation/examples/src*.

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 131
- ◆ *Managing Input and Output* on page 141
- ◆ *Licensing an Application* on page 153

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 131
- ◆ *Using the Interactive Optimizer for Debugging* on page 136
- ◆ *Eliminating Common Programming Errors* on page 138

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* on page 132
- ◆ *Identify Routines to Use* on page 132
- ◆ *Test Interactively* on page 132
- ◆ *Assemble Data Efficiently* on page 132

- ◆ *Test Data* on page 133
 - ◆ *Choose an Optimizer* on page 133
 - ◆ *Program with a View toward Maintenance and Modifications* on page 134
-

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 numeric 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 `CPXcheckData` 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:

- ◆ Use the primal simplex, dual simplex, or primal-dual barrier optimizers to solve linear and quadratic programs.
- ◆ Use the barrier optimizer to solve quadratically constrained programming problems.
- ◆ The network optimizer is appropriate for solving linear and quadratic programs with large embedded networks.

- ◆ Use the MIP optimizer 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 *Solving LPs: Simplex Optimizers* on page 161 and *Solving Problems with a Quadratic Objective (QP)* on page 217 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 *Solving Mixed Integer Programming Problems (MIP)* on page 245.

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 131 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 customer 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 a natural language, such as English.

Avoid Side-Effects

It is a 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 lefthand 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 118, 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 Concert or Callable Library application.

Use one of these methods, specifying a file type of SAV, to create a binary representation of the problem object from your application in a SAV file.

- ◆ `IloCplex::exportModel`
- ◆ `IloCplex.exportModel`
- ◆ `Cplex.ExportModel`
- ◆ `CPXwriteprob`

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.

With respect to parameter settings, you can write a parameter file with the file extension `.prm` from your application by means of one of these methods:

- ◆ `IloCplex::writeParam` in the C++ API
- ◆ `IloCplex.writeParam` in the Java API
- ◆ `Cplex.WriteParam` in the .NET API
- ◆ `CPXwriteparam` in the Callable Library
- ◆ `write file.prm` in the Interactive Optimizer

The Interactive Optimizer can read a `.prm` file and then set parameters exactly as they are in your application.

In the other direction, you can use the `display` command in the Interactive Optimizer to show the nondefault parameter settings; you can then save those settings in a `.prm` file for re-use later. See the topic *Saving a Parameter Specification File* on page 17 in the reference manual of the Interactive Optimizer for more detail about using a parameter file in this way.

To use the Interactive Optimizer for debugging, you first need to write a version of the problem from the application into a formatted file that can then be loaded into the Interactive Optimizer. To do so, insert a call to the method `IloCplex::exportModel` or to the routine `CPXwriteprob` into your application. Use that call to create a file, whether an LP, SAV, or MPS formatted problem file. (*Understanding File Formats* on page 142 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 customer 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 righthand 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 139
- ◆ *Clean House and Try Again* on page 139
- ◆ *Read Your Messages* on page 139
- ◆ *Check Return Values* on page 139
- ◆ *Beware of Numbering Conventions* on page 139

- ◆ *Make Local Variables Temporarily Global* on page 140
 - ◆ *Solve the Problem You Intended* on page 140
 - ◆ *Special Considerations for Fortran* on page 140
 - ◆ *Tell Us* on page 140
-

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 136, and the diagnostic routines, as in *Using Diagnostic Routines for Debugging* on page 118.

You should not ignore any ILOG CPLEX warning message in this situation either, so read your messages, as in *Read Your Messages* on page 139.

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.

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 142;
- ◆ *Using Concert XML Extensions* on page 145
- ◆ *Using Concert csvReader* on page 146;
- ◆ *Managing Log Files* on page 147;
- ◆ *Controlling Message Channels* on page 148.

Note: *There are platforms that 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 142;
- ◆ *Working with MPS Files* on page 143;
- ◆ *Converting File Formats* on page 144.

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:

$$\begin{array}{l} \text{Maximize} \qquad \qquad \qquad 2x_2 \quad + \quad 3x_3 \\ \text{subject to} \\ \qquad \qquad \qquad -x_1 \quad + \quad x_2 \quad + \quad x_3 \leq 20 \\ \qquad \qquad \qquad x_1 \quad - \quad 3x_2 \quad + \quad x_3 \leq 30 \\ \text{with these bounds} \\ \qquad \qquad \qquad 0 \leq x_1 \leq 40 \\ \qquad \qquad \qquad 0 \leq x_2 \leq +\infty \\ \qquad \qquad \qquad 0 \leq x_3 \leq +\infty \end{array}$$

Since ILOG CPLEX reads the objective function as the first row, the two columns appearing there will become the first two variables. When the problem is displayed or rewritten into

another LP file, the variables there will appear in a different order within each row. In this example, if you execute the command `display problem all`, you will see this:

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

That is, x_1 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 files created by other modeling systems that respect the MPS format. 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 righthand 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 numeric 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 righthand side names);
- ◆ extraneous, uncommented characters;

- ◆ blanks in lieu of repeated name fields, such as bound vector names and righthand 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, 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
-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 transformed 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 *ILOG CPLEX C++ API 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 `plex.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 managing a log file:

- ◆ *Creating, Renaming, Relocating Log Files* on page 147;
- ◆ *Closing Log Files* on page 147.

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. In the C++ and Java APIs, the class `IloCplex` inherits methods from the Concert Technology class `IloAlgorithm`, methods that enable you to control input and output channels.

The following sections offer more details about these ideas:

- ◆ *Parameter for Output Channels* on page 148;
- ◆ *Callable Library Routines for Message Channels* on page 149;
- ◆ *Example: Callable Library Message Channels* on page 150;
- ◆ *Concert Technology Message Channels* on page 152.

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
<code>dialog</code>	messages related to interactive use; e.g., prompts, help messages, greetings
<code>errors</code>	messages to inform user that operation could not be performed and why
<code>logonly</code>	message to record only in file (not on screen) e.g., multiline messages
<code>results</code>	information explicitly requested by user; state, change, progress information
<code>warnings</code>	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
<code>dialog</code>	<code>y</code>	blank directs output to screen but not to a file
<code>errors</code>	<code>y</code>	blank directs output to screen and to a file
<code>logonly</code>	<code>n</code>	blank directs output only to a file, not to screen
<code>results</code>	<code>y</code>	blank directs output to screen and to a file
<code>warnings</code>	<code>y</code>	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.

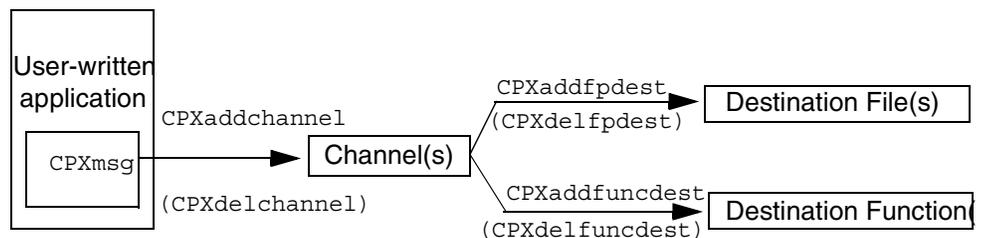


Figure 6.1 ILOG CPLEX Message Handling Routines

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 to 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.

The complete program `lpex5.c` appears online in the standard distribution at `yourCPLEXinstallation/examples/src`.

Concert Technology Message Channels

In the C++ API of Concert Technology, the class `IloEnv` initializes output streams for general information, for error messages, and for warnings. The class `IloAlgorithm` supports these communication streams, and the class `IloCplex` inherits its methods. For general output, there is the method `IloAlgorithm::out`. For warnings and nonfatal conditions, there is the method `IloAlgorithm::warning`. For errors, there is the method `IloAlgorithm::error`.

By default, an instance of `IloEnv` defines the output stream referenced by the method `out` as the system `cout` in the C++ API, but you can use the method `setOut` to redefine it as you prefer. For example, to suppress output to the screen in a C++ application, use this method with this argument:

```
setOut(IloEnv::getNullStream)
```

Likewise, you can use the methods `IloAlgorithm::setWarning` and `setError` to redefine those channels as you prefer.

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 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.

A runtime license 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 154
- ◆ *Routines and Methods Used for Licensing* on page 154
- ◆ *Examples* on page 155
- ◆ *Summary* on page 158

Types of ILM Runtime Licenses

ILM runtime licenses come in two forms: file-based and memory-based. The following sections document those two forms of license.

- ◆ *File-Based RTNODE, RTSTOKEN or TOKEN Keys* on page 154
- ◆ *Memory-Based RUNTIME Keys* on page 154

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 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 .NET API of Concert Technology

These file-based keys 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 .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 .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 .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 .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 .NET API Reference Manual*.

Examples

Here are some code samples that illustrate the use of those runtime license routines and methods.

- ◆ *CPXputenv Routine for C and C++ Users* on page 155
- ◆ *The putenv Method for Java Users* on page 156
- ◆ *The Putenv Method for .NET Users* on page 157
- ◆ *CPXRegisterLicense Routine for C and C++ Users* on page 157
- ◆ *The registerLicense Method for Java Users* on page 157
- ◆ *The RegisterLicense Method for .NET Users* on page 158

CPXputenv Routine for C and C++ Users

This example illustrates the routine `CPXputenv` to open the CPLEX environment

```
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);
.

```

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.

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);
}

```

The Putenv Method for .NET Users

Here is an example using Concert Technology for .NET users:

```
Cplex.Putenv("ILOG_LICENSE_FILE=../../../certify/access.e.ilm");
try {
    cplex = new Cplex();
}
catch (ILOG.Concert.Exception e) {
    System.Console.WriteLine("Concert exception caught: " + e);
}
```

CPXRegisterLicense Routine for C and C++ Users

The following is an example showing how to use the routine CPXRegisterLicense.

```
static char *ilm_license=\
"LICENSE ILOG Incline\n\
  RUNTIME CPLEX 9.200 21-Jul-2005 R81GM34ECZTS N , 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 9.200 021-Jul-2005 R81GM34ECZTS N ,
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);
    }
}

```

The RegisterLicense Method for .NET Users

Here is an example for .NET users applying `Cplex.RegisterLicense`:

```

internal static string ilm_CPLEX_license="LICENSE ILOG User\n RUNTIME CPLEX
9.200 05-Aug-2005 62RAR21A8NC5 N any , options: m ";
internal static int ilm_CPLEX_license_signature=863909;
public static void Main(string[] args) {
    try {
        Cplex.RegisterLicense(ilm_CPLEX_license, ilm_CPLEX_license_signature);
        Cplex cplex = new Cplex();
    }
    catch (ILOG.Concert.Exception e) {
        System.Console.WriteLine("Expected Concert exception caught: " + e);
    }
}

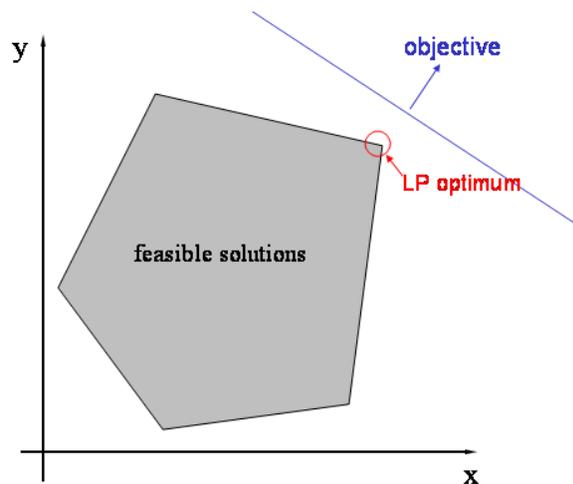
```

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* 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 161
- ◆ *Solving LPs: Barrier Optimizer* on page 187
- ◆ *Solving Network-Flow Problems* on page 207
- ◆ *Solving Problems with a Quadratic Objective (QP)* on page 217
- ◆ *Solving Problems with Quadratic Constraints (QCP)* on page 229

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 the LP optimizers of ILOG CPLEX. This chapter emphasizes primal and dual simplex optimizers. It contains sections about:

- ◆ *Choosing an Optimizer for Your LP Problem* on page 162
- ◆ *Tuning LP Performance* on page 165
- ◆ *Diagnosing Performance Problems* on page 173
- ◆ *Diagnosing LP Infeasibility* on page 179
- ◆ *Example: Using a Starting Basis in an LP Problem* on page 184

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 `RootAlg` in the C++, Java, and .NET APIs, `CPX_PARAM_LPMETHOD` in the Callable Library, and named `lpmethod` in the Interactive Optimizer. In Concert Technology, 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 Technology)
- ◆ `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	<i>Automatic Selection of Optimizer</i> on page 163
1	Primal Simplex	<i>Primal Simplex Optimizer</i> on page 164
2	Dual Simplex	<i>Dual Simplex Optimizer</i> on page 163
3	Network Simplex	<i>Network Optimizer</i> on page 164
4	Barrier	<i>Barrier Optimizer</i> on page 164
5	Sifting	<i>Sifting Optimizer</i> on page 164
6	Concurrent Dual, Barrier, and Primal	<i>Concurrent Optimizer</i> on page 165

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. The dual simplex method is the first choice for optimizing a linear programming problem, especially for primal-degenerate problems with little variability in the righthand 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. The primal simplex method is not 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 numeric performance in both primal and dual form. Consequently, if you have a problem where numeric 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. *Solving Network-Flow Problems* on page 207 explores this optimizer in greater detail.

Barrier Optimizer

The barrier optimizer offers an approach particularly efficient on large, sparse problems (for example, more than 100 000 rows or columns, and no more than perhaps a dozen nonzeros per column) and sometimes on other models as well. The barrier optimizer is sufficiently different in nature from the other optimizers that it is discussed in detail in *Solving LPs: Barrier Optimizer* on page 187.

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 in 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 *Parallel Optimizers* on page 447, especially *Concurrent Optimizer* on page 452.

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 162. The following sections suggest other features of ILOG CPLEX to consider in tuning the performance of your application:

- ◆ *Preprocessing* on page 166
- ◆ *Starting from an Advanced Basis* on page 168
- ◆ *Simplex Parameters* on page 169

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 (default)
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 turn on the `MemoryEmphasis` (bool) (`CPX_PARAM_MEMORYEMPHASIS` (int) in the Callable Library) parameter to conserve memory, 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 compression can make more memory available for the optimizer that the user has called. To conserve memory, you can also turn on the `MemoryEmphasis` (bool) (`CPX_PARAM_MEMORYEMPHASIS` (int) in the Callable Library) parameter.

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 *Solving LPs: Barrier Optimizer* on page 187). The simplex tuning suggestions appear in the following topics:

- ◆ *Pricing Algorithm and Gradient Parameters* on page 169
- ◆ *Scaling* on page 171
- ◆ *Crash* on page 172
- ◆ *Memory Management and Problem Growth* on page 172

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 `cpplex.log`. *Managing Log Files* on page 147 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 scaling indicator parameter (`ScaInd` in Concert Technology, `CPX_PARAM_SCAIND` in the Callable Library). The scaling indicator may be set as in Table 8.6.

Table 8.6 *ScaIndParameter Settings for Scaling Methods*

ScalInd 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.

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 numeric 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.

An alternative to obtaining more memory is to conserve memory that is available. The memory emphasis parameter can help you in this respect.

- ◆ `MemoryEmphasis` (bool) in Concert Technology
- ◆ `CPX_PARAM_MEMORYEMPHASIS` (int) in the Callable Library
- ◆ `emphasis_memory` in the Interactive Optimizer

If you set the memory emphasis parameter to its optional value of 1 (one), then ILOG CPLEX will adopt memory conservation tactics at the beginning of optimization rather than only after the shortage becomes apparent. These tactics may still have a noticeable impact on solution speed because these tactics change the emphasis from speed to memory utilization, but they could give an improvement over the default in the case where memory is insufficient.

The following sections offer further guidance about memory conservation if memory emphasis alone does not do enough for your problem.

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 (that is, factoring more often) 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.

Numeric Difficulties

ILOG CPLEX is designed to handle the numeric difficulties of linear programming automatically. In this context, numeric 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 numeric precision of your computer.

Numerical Emphasis Settings

The numerical emphasis parameter controls the degree of numerical caution used during optimization of a model.

- ◆ `NumericalEmphasis` (bool) in Concert Technology
- ◆ `CPX_PARAM_NUMERICAL EMPHASIS` (int) in the Callable Library
- ◆ `emphasis numerical` in the Interactive Optimizer

At its default setting, ILOG CPLEX employs ordinary caution in dealing with the numerical properties of the computations it must perform. Under the optional setting, ILOG CPLEX uses extreme caution.

This emphasis parameter is different in style from the various tolerance parameters in ILOG CPLEX. The purpose of the emphasis parameter is to relieve the user of the need to analyze which tolerances or other algorithmic controls to try. Instead, the user tells ILOG CPLEX that the model about to be solved is known to be susceptible to unstable numerical behavior and lets ILOG CPLEX make the decisions about how best to proceed.

There may be a tradeoff between solution speed and numerical caution. You should not be surprised if your model solves less rapidly at the optional setting of this parameter, because each iteration may potentially be noticeably slower than at the default. On the other hand, if the numerical difficulty has been causing the optimizer to proceed less directly to the optimal solution, it is possible that the optional setting will reduce the number of iterations, thus leading to faster solution. When the user chooses an emphasis on extreme numerical caution, solution speed is in effect treated as no longer the primary emphasis.

Numerically Sensitive Data

There is no absolute link between the form of data in a model and the numeric 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 $\frac{1}{3}$ as .3333333 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 $\frac{1}{3} x_1 + \frac{2}{3} x_2 = 1$. In perfect arithmetic, it is equivalent to $x_1 + 2 x_2 = 3$. However, if you express the fractional form with decimal data values, some truncation is unavoidable. If you happen to include the truncated decimal form of the constraint in the same model with some differently-truncated form or even the exact-integer data form, an unexpected result could easily occur. Consider the following problem formulation:

```

Maximize
  obj: x1 + x2
Subject To
  c1: 0.3333333 x1 + 0.666667 x2 = 1
  c2: x1 + 2 x2 = 3
End

```

With default numeric tolerances, this will deliver an optimal solution of $x_1=1.0$ and $x_2=1.0$, 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: 0.3333333333 x1 + 0.6666666667 x2 = 1
  c2: x1 + 2 x2 = 3
End

```

The solution to this problem has $x_1=3.0$ and $x_2=0.0$, 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 numeric 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 form of the constraint is not present with the decimal form, the truncated decimal form 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 situation would arise from using type `float` instead of `double`. Such data will be accurate only to about 8 decimal digits, so that (for example) 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 (counting from the right) 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 171 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 179 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 the perturbed problem may differ too greatly from your original problem. In such a case, consider reducing the value of the perturbation constant (ϵ_{Per} in Concert Technology, `CPX_PARAM_EPPER` in the Callable Library).

Inability to Stay Feasible

If a problem repeatedly becomes infeasible in Phase II (that is, after ILOG CPLEX has achieved a feasible solution), then numeric difficulties may be occurring. It may help to increase the Markowitz tolerance in such a case. By default, the value of the parameter ϵ_{Mrk} 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 numeric 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 numeric 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 conflict refiner and FeasOpt:

- ◆ The conflict refiner is invoked by the routine `CPXrefineconflict` in the Callable Library or by the method `refineConflict` in Concert Technology. It finds a set of conflicting constraints and bounds in a model and refines the set to be minimal in a sense

that you declare. It then reports its findings for you to take action to repair that conflict in your infeasible model. For more about this feature, see *Diagnosing Infeasibility by Refining Conflicts* on page 353.

- ◆ 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 *Repairing Infeasibility: FeasOpt* on page 183.

With the help of those tools, you may be able to modify your problem to avoid infeasibility.

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 182, or after you reduce feasibility tolerance, as explained in *Maximum Bound Infeasibility: Identifying Largest Bound Violation* on page 182. 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 177. 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 Quality

Infeasibility and unboundedness in linear programs are closely related. When the linear program ILOG CPLEX solves is *infeasible*, the associated dual linear program has an *unbounded ray*. Similarly, when the dual linear program is *infeasible*, the primal linear program has an *unbounded ray*. This relationship is important for proper interpretation of infeasible solution output.

The treatment of models that are unbounded involves a few subtleties. Specifically, a declaration of unboundedness means that ILOG CPLEX has determined that the model has an unbounded ray. Given any feasible solution x with objective z , a multiple of the unbounded ray can be added to x to give a feasible solution with objective $z-1$ (or $z+1$ for maximization models). Thus, if a feasible solution exists, then the optimal objective is unbounded. Note that ILOG CPLEX has **not** necessarily concluded that a feasible solution exists. To determine whether ILOG CPLEX has also concluded that the model has a feasible solution, use one of these routines or methods:

- ◆ `CPXsolninfo` in the Callable Library
- ◆ `isPrimalFeasible` or `isDualFeasible` of the class `IloCplex` in Concert Technology

By default, individual infeasibilities are written to a log file but not displayed on the screen. To display the infeasibilities on your screen, in Concert Technology, use methods of the environment to direct the output stream to a log file; in the Interactive Optimizer, use the command `set output logonly y cplex.log`.

For C++ applications, see *Accessing Solution Information* on page 55, and for Java applications, see *Accessing Solution Information* on page 80. Those sections highlight the application programming details of how to retrieve statistics about the quality of a solution.

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.

When the simplex optimizer detects infeasibility in the primal or dual linear program (LP), parts of the solution it provides are relative to the Phase I linear program it solved to conclude infeasibility. In other words, the result you see in such a case is **not** the solution

values computed relative to the original objective or original righthand side vector. Keep this distinction in mind when you interpret solution quality; otherwise, you may be surprised by the results. In particular, when ILOG CPLEX determines a linear program is infeasible using the primal simplex method, the reduced costs and dual variables provided in the solution are relative to the objective of the Phase I linear program it solved. Similarly, when ILOG CPLEX determines a linear program is unbounded because the dual simplex method detected dual infeasibility, the primal and slack variables provided in the solution are relative to the Phase I linear program created for the dual simplex optimizer.

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 parameter for feasibility tolerance (ϵ_{PRHS} in Concert Technology, `CPX_PARAM_EPRHS` in the Callable Library).

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 for optimality tolerance (ϵ_{Opt} in Concert Technology, `CPX_PARAM_EPOPT` in the Callable Library).

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 numeric 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 180 outlines steps to follow in this situation.

Finding a Conflict

If ILOG CPLEX reports that your problem is infeasible, then you can invoke tools of ILOG CPLEX to help you analyze the source of the infeasibility. These diagnostic tools compute a set of conflicting constraints and column bounds that would be feasible if one of them (a constraint or variable) were removed. Such a set is known as a *conflict*. For more about detecting conflicts, see *Diagnosing Infeasibility by Refining Conflicts* on page 353.

Repairing 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.

In contrast, an automated approach offered in ILOG CPLEX is known as FeasOpt (for feasible optimization). FeasOpt attempts to repair an infeasibility by modifying the model according to preferences set by the user. For more about this approach, see *Repairing Infeasibilities with FeasOpt* on page 371

Example: Using a Starting Basis in an LP Problem

Here is an approach mentioned in the section *Tuning LP Performance*. The approach is to start with an advanced basis. The following small example in C++ and in C demonstrates an approach to setting a starting basis by hand. *Example ilolpex6.cpp* on page 184 is from Concert Technology in the C++ API. *Example lpex6.c* on page 184 is from the Callable Library in C.

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.

The complete program `ilolpex6.cpp` appears online in the standard distribution at `yourCPLEXinstallation/examples/src`.

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. After 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.

The complete program `lpex6.c` appears online in the standard distribution at `yourCPLEXinstallation/examples/src`.

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:

- ◆ *Introducing the Barrier Optimizer* on page 188
- ◆ *Using the Barrier Optimizer* on page 190
- ◆ *Special Options* on page 191
- ◆ *Controlling Crossover* on page 191
- ◆ *Using SOL File Format* on page 192
- ◆ *Interpreting the Barrier Log File* on page 192
- ◆ *Understanding Solution Quality from the Barrier LP Optimizer* on page 195
- ◆ *Tuning Barrier Optimizer Performance* on page 197
- ◆ *Overcoming Numeric Difficulties* on page 201
- ◆ *Diagnosing Infeasibility Reported by Barrier Optimizer* on page 206

Introducing the Barrier Optimizer

The ILOG CPLEX Barrier Optimizer is well suited to large, sparse problems. An alternative to the simplex optimizers, which are also suitable to problems in which the matrix representation is dense, the barrier optimizer 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 $\mathbf{c}^T \mathbf{x}$

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

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

where \mathbf{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 $\mathbf{b}^T \mathbf{y} - \mathbf{u}^T \mathbf{w} + \mathbf{l}^T \mathbf{z}$

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

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 ($\mathbf{A}\mathbf{x} = \mathbf{b}$, $x + s = u$) and dual constraints ($\mathbf{A}^T \mathbf{y} + \mathbf{z} - \mathbf{w} = \mathbf{c}$) 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 $\mathbf{A}\mathbf{A}^T$ 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 100 000 rows or columns. It is not *always* the best choice, though, for sparse models with more than 100 000 rows. 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 (perhaps no more than a dozen nonzero values per column).

In short, denseness or sparsity are not the deciding issues when you are deciding whether to use the barrier optimizer. In fact, 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, not simply at denseness, sparseness, or problem size. (How to check those characteristics is explained later in this chapter in *Cholesky Factor in the Log File* on page 194, and *Nonzeros in Lower Triangle of AA^T in the Log File* on page 193).

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 numeric difficulties). Optionally, you can also execute barrier optimization with a dual crossover or with no crossover at all. The section *Controlling Crossover* on page 191 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.
- ◆ By default, the barrier optimizer crossover to produce a basis. However, you may choose to run the barrier optimizer without crossover. In such a case, the fact that barrier without crossover does not produce a basic solution has 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 *numeric 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. In these cases, concurrent optimization, as documented in *Concurrent Optimizer* on page 452, may be helpful.
- ◆ 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. Again, concurrent optimization, as documented in *Concurrent Optimizer* on page 452, may be helpful.

Using the Barrier Optimizer

As you have read in *Introducing the Barrier Optimizer* on page 188, 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
<code>IloCplex::Barrier</code>	Concert Technology for C++ users
<code>IloCplex.Algorithm.Barrier</code>	Concert Technology for Java users
<code>Cplex.Algorithm.Barrier</code>	Concert Technology for .NET users
<code>CPX_ALG_BARRIER</code>	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
<code>cplex.solve</code>	Concert Technology for C++ users
<code>cplex.solve</code>	Concert Technology for Java users
<code>Cplex.Solve</code>	Concert Technology for .NET users
<code>CPXlpopt</code>	Callable Library
<code>optimize</code>	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 SOL 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 SOL-format file (that is, a solution file with the extension `.sol`). You can then read that solution file into ILOG CPLEX before you initiate a crossover at a later time. After you read a solution file into ILOG CPLEX, all three optimizers (primal simplex, dual simplex, and barrier simplex) automatically invoke crossover. See the *ILOG CPLEX File Format Reference Manual*, especially *SOL File Format: Solution Files* on page 38, for more about solution files.

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 `plex.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
```

The opening lines of that log file record information about *preprocessing* by the ILOG CPLEX presolver and aggregator. After those preprocessing statistics, the next line records the number of *nonzeros in the lower triangle* of a particular matrix, AA^T , denoted $A*A'$ in the log file.

Nonzeros in Lower Triangle of AA^T in the Log File

The number of nonzeros in the lower triangle of AA^T gives an early indication of how long each barrier iteration will take in terms of a relative measure of time. 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 of the reduced LP, 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.

In the Interactive Optimizer, you can examine the histogram of column counts with the command `display problem histogram`.

Ordering-Algorithm Time in the Log File

After the number of nonzeros in the lower triangle of AA^T , ILOG CPLEX records the time required by the ordering algorithm. (The ILOG CPLEX Barrier Optimizer offers you a choice of several ordering algorithms, explained in *Choosing an Ordering Algorithm* on page 200.) 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 AA^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 relative time that will be required to perform each iteration of the barrier optimizer.

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 $Ax = b$ and $x + s = u$, and the dual constraints are written as $A^T y + z - w = c$. 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 - Ax $
upper	Upper Inf	$ u - (x + s) $
dual	Dual Inf	$ c - yA - 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 $c^T x$
dual objective	dual objective value $b^T y - u^T w + l^T z$
duality gap	difference between primal and dual objectives
complementarity	sum of column and row complementarity

Table 9.6 Barrier Solution Quality Display (Continued)

Item	Meaning
column complementarity (total)	sum of $ (x_j - l_j) \bullet z_j + (u_j - x_j) \bullet w_j $
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'p + 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'p + 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 p_i 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 Ax on each row and $y^T A$ on each column. In the primal case, for each row, consider the nonzero coefficients and the x_j values used to compute Ax . 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 numeric difficulties. The following sections document these parameters:

- ◆ *Memory Emphasis: Letting the Optimizer Use Disk for Storage* on page 198
- ◆ *Preprocessing* on page 199;
- ◆ *Detecting and Eliminating Dense Columns* on page 200;
- ◆ *Choosing an Ordering Algorithm* on page 200;
- ◆ *Using a Starting-Point Heuristic* on page 201.

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 194.

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 193. *Detecting and Eliminating Dense Columns* on page 200 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 *Saving a Parameter Specification File* on page 17 in the reference manual *ILOG CPLEX Interactive Optimizer Commands*.

Memory Emphasis: Letting the Optimizer Use Disk for Storage

At default settings, the ILOG CPLEX barrier optimizer will do all of its work in central memory (also variously referred to as RAM, core, or physical memory). For models too large to solve in the central memory on your computer, or in cases where you simply do not want 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 access to disk is slower than access to central memory, there may be some lost performance by this choice on models that could be solved entirely in central memory, but the out-of-core feature in the barrier optimizer is designed to make this trade-off as efficient as possible. It generally will be far more effective than relying on the virtual memory (that is, the swap space) of your operating system.

To activate the out-of-core feature, set the memory emphasis parameter to 1 (one) instead of its default value of 0 (zero).

- ◆ `MemoryEmphasis` (bool) in Concert Technology
- ◆ `CPX_PARAM_MEMORYEMPHASIS` (int) in the Callable Library
- ◆ `emphasis memory` in the Interactive Optimizer

This memory emphasis feature will also invoke other memory conservation tactics, such as compression of the data within presolve.

Memory emphasis uses some working memory in RAM to store the portion of the factor on which it is currently performing computation. You can improve performance by allocating more working memory by means of the working memory parameter.

- ◆ `WorkMem` in Concert Technology
- ◆ `CPX_PARAM_WORKMEM` in the Callable Library
- ◆ `workmem` in the Interactive Optimizer

More working memory allows the optimizer to transfer less data to and from disk. In fact, the Cholesky factor matrix will not be written to disk at all if its size does not exceed the value of the working memory parameter. The default for this parameter is 128 megabytes.

When the barrier optimizer operates with memory emphasis, the location of disk storage is controlled by the working directory parameter.

- ◆ `WorkDir` in Concert Technology
- ◆ `CPX_PARAM_WORKDIR` in the Callable Library
- ◆ `workdir` in the Interactive Optimizer

For example, to use the directory `/tmp/mywork`, set the working directory parameter to the string `/tmp/mywork`. The value of the working directory parameter should be specified as the name of a directory that already exists, and ILOG CPLEX will create its working directory as a subdirectory there. At the end of barrier optimization, ILOG CPLEX will automatically delete any working directories it created, leaving the directory specified by the working directory parameter intact.

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, including MIQP and MIQCP.

Detecting and Eliminating Dense Columns

Dense columns can significantly degrade the performance of the barrier optimizer. A dense column is one in which a given variable appears in many rows. So that you can detect dense columns, 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`.

Nonzeros in Lower Triangle of AA^T in the Log File on page 193 explains how to examine a log file from the barrier optimizer in order to tell which columns CPLEX detects as dense at its current settings.

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 (according to the log file), 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 194, 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 `BarStartAlg` or `CPX_PARAM_BARSTARTALG`.

Overcoming Numeric Difficulties

As noted in *Differences between Barrier and Simplex Optimizers* on page 189, the algorithms in the barrier optimizer have very different numeric properties from those in the

simplex optimizer. While the barrier optimizer is often extremely fast, particularly on very large problems, numeric 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.

Numerical Emphasis Settings

Before you try tactics that apply to specific symptoms, as described in the following sections, a useful ILOG CPLEX parameter to try is the numerical emphasis parameter.

- ◆ `NumericalEmphasis` (bool) in Concert Technology
- ◆ `CPX_PARAM_NUMERICALEMPHASIS` (int) in the Callable Library
- ◆ `emphasis numerical` in the Interactive Optimizer

Unlike the following suggestions, which deal with knowledge of the way the barrier optimizer works or with details of your specific model, this parameter is intended as a way to tell ILOG CPLEX to exercise more than the usual caution in its computations. When you set it to its nondefault value specifying extreme numerical caution, various tactics are invoked internally to try to avoid loss of numerical accuracy in the steps of the barrier algorithm.

Be aware that the nondefault setting may result in slower solution times than usual. The effect of this setting is to shift the emphasis away from fastest solution time and toward numerical caution. On the other hand, if numerical difficulty is causing the barrier algorithm to perform excessive numbers of iterations due to loss of significant digits, it is possible that the setting of extreme numerical caution could actually result in somewhat faster solution times. Overall, it is difficult to project the impact on speed when using this setting.

The purpose of this parameter setting is **not** to generate "more accurate solutions" particularly where the input data is in some sense unsatisfactory or inaccurate. The numerical caution is applied during the steps taken by the barrier algorithm during its convergence toward the optimum, to help it do its job better. On some models, it may turn out that solution quality measures are improved (Ax-b residuals, variable-bound violations, dual values, and so forth) when ILOG CPLEX exercises numerical caution, but this would be a secondary outcome from better convergence.

Difficulties in the Quality of Solution

Understanding Solution Quality from the Barrier LP Optimizer on page 195 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 197 and recapitulated here in the following sections.

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 numeric difficulties, they may eliminate the numeric 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 192, 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 (zero) may improve numeric 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 201, the default starting-point heuristic works well for most problems suitable to barrier optimization. But for a model that is exhibiting numeric 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 201.

Difficulties during Optimization

Numeric difficulties can degrade performance of the ILOG CPLEX Barrier Optimizer or even prevent convergence toward a solution. There are several possible sources of numeric difficulties:

- ◆ elimination of too many dense columns may cause numeric instability;
- ◆ tight convergence tolerance may aggravate small numeric inconsistencies in a problem;
- ◆ unbounded optimal faces may remain undetected and thus prevent convergence.

The following subsections offer guidance about overcoming those difficulties.

Numeric Instability Due to Elimination of Too Many Dense Columns

Detecting and Eliminating Dense Columns on page 200 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 numeric instability.

You can check how many dense columns ILOG CPLEX removes by looking at the preprocessing statistics at the beginning of the log file. For example, the following log file shows that CPLEX removed 2249 columns, of which nine were dense.

```
Selected objective sense: MINIMIZE
Selected objective name: obj
Selected RHS name: rhs
Selected bound name: bnd

Problem 'XXX.mps' read.
Read time = 0.03 sec.
Tried aggregator 1 time.
LP Presolve eliminated 2200 rows and 2249 columns.
Aggregator did 8 substitutions.
Reduced LP has 171 rows, 182 columns, and 1077 nonzeros.
Presolve time = 0.02 sec.

***NOTE: Found 9 dense columns.

Number of nonzeros in lower triangle of A*A' = 6071
Using Approximate Minimum Degree ordering
Total time for automatic ordering = 0.00 sec.
Summary statistics for Cholesky factor:
  Rows in Factor           = 180
  Integer space required   = 313
  Total non-zeros in factor = 7286
  Total FP ops to factor   = 416448
```

If you observe that the removal of too many dense columns results in numeric 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 Numeric Inconsistencies and Tight Convergence Tolerance

If your problem contains small numeric 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 numeric 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 Infeasibility Reported by Barrier Optimizer

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, one approach to overcoming the infeasibility is to invoke `FeasOpt` or the conflict refiner. See *Repairing Infeasibilities with FeasOpt* on page 371 and *Diagnosing Infeasibility by Refining Conflicts* on page 353 for an explanation of these tools.

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.

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 208
- ◆ *Formulating a Network Problem* on page 208
- ◆ *Example: Network Optimizer in the Interactive Optimizer* on page 209
- ◆ *Example: Using the Network Optimizer with the Callable Library `netex1.c`* on page 213
- ◆ *Solving Network-Flow Problems as LP Problems* on page 214
- ◆ *Example: Network to LP Transformation `netex2.c`* on page 216
- ◆ *Solving Problems with the Network Optimizer* on page 211

Choosing an Optimizer: Network Considerations

As explained in *Using the Callable Library in an Application* on page 109, 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 Callable 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:

$$\text{Minimize (or maximize)} \quad \sum_{a \in A} (c_a x_a)$$

$$\text{subject to} \quad \sum_{a \in T_n} x_a - \sum_{a \in H_n} x_a = s_n \quad \forall (n \in N)$$

$$\text{with these bounds} \quad l_a \leq x_a \leq u_a \quad \forall (a \in A)$$

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.

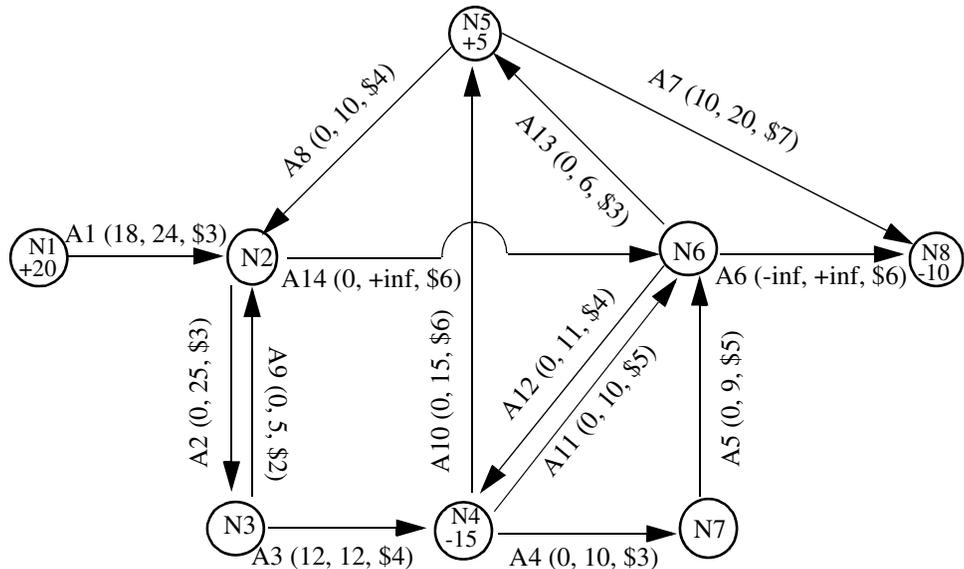


Figure 10.1 A Directed Network with Arc-Capacity, Flow-Cost, Sinks, and Sources

The example in Figure 10.1 corresponds to the results of running the `netex1.c`. If you run that application, it will produce a file named `netex1.net` which can be read into the Interactive Optimizer with the command `read netex1.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 (150)

Network - Optimal:  Objective = 2.690000000000e+002
Solution time =    0.01 sec.  Iterations = 9 (9)
```

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 (that is, problems with a positive semi-definite quadratic term in the objective function), but **not** when solving quadratically constrained problems. To do so using the Callable Library, you set the parameter `CPX_PARAM_QPMETHOD` 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 210.

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 109, `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`.

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 online in the standard distribution at [yourCPLEXinstallation/examples/src](#).

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 210 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

$$a_1 = 20$$

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`). You can find the data of this example in the file `examples/data/infnet.net`. After reading the data from that file, the example then attempts to solve the problem by calling the Callable Library routine `CPXNETprimopt`. If it determines that the problem is infeasible, it then invokes the conflict refiner to analyze the problem and possibly indicate the cause of the infeasibility.

The complete program `netex2.c` appears online in the standard distribution at *yourCPLEXinstallation/examples/src*.

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 218
- ◆ *Entering QPs* on page 219
- ◆ *Saving QP Problems* on page 222
- ◆ *Changing Problem Type in QPs* on page 222
- ◆ *Changing Quadratic Terms* on page 223
- ◆ *Optimizing QPs* on page 224
- ◆ *Example: Creating a QP, Optimizing, Finding a Solution* on page 226
- ◆ *Example: Reading a QP from a File qpex2.c* on page 228

Identifying Convex QPs

Conventionally, a quadratic program (QP) is formulated this way:

$$\text{Minimize } \frac{1}{2} \mathbf{x}^T \mathbf{Q} \mathbf{x} + \mathbf{c}^T \mathbf{x}$$

subject to $\mathbf{A} \mathbf{x} \sim \mathbf{b}$

with these bounds $l \leq \mathbf{x} \leq u$

where the relation \sim may be any combination of equal to, less than or equal to, greater than or equal to, or range constraints. As in other problem formulations, l indicates lower and u upper bounds. \mathbf{Q} is a matrix of objective function coefficients. That is, the elements Q_{jj} are the coefficients of the quadratic terms x_j^2 , and the elements Q_{ij} and Q_{ji} are summed together to be the coefficient of the term $x_i x_j$.

ILOG CPLEX distinguishes two kinds of \mathbf{Q} matrices:

- ◆ In a *separable* problem, only the diagonal terms of the matrix are defined.
- ◆ In a *nonseparable* problem, at least one off-diagonal term of the matrix is nonzero.

ILOG CPLEX 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 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.

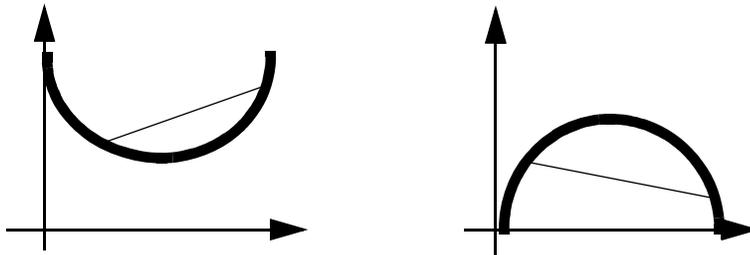


Figure 11.1 Minimize a Convex Objective Function, Maximize a Concave Objective Function

In formal terms, the question of whether a quadratic objective function is convex or concave is equivalent to whether the matrix \mathbf{Q} is positive semi-definite or negative semi-definite. For

convex QPs, \mathbf{Q} must be *positive semi-definite*; that is, $\mathbf{x}^T\mathbf{Q}\mathbf{x} \geq 0$ for every vector \mathbf{x} , whether or not \mathbf{x} is feasible. For concave maximization problems, the requirement is that \mathbf{Q} must be *negative semi-definite*; that is, $\mathbf{x}^T\mathbf{Q}\mathbf{x} \leq 0$ for every vector \mathbf{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 \mathbf{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 \mathbf{Q} are of the correct sign. For a nonseparable case, it may be less easy to determine in advance the convexity of \mathbf{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 37 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.

- ◆ *Matrix View* on page 219
- ◆ *Algebraic View* on page 220
- ◆ *Examples for Entering QPs* on page 220
- ◆ *Reformulating QPs to Save Memory* on page 221

Matrix View

In the matrix view, commonly found in textbook presentations of QP, the objective function is defined as $1/2 \mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x}$, where \mathbf{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 \mathbf{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 \mathbf{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.

This symmetry restriction applies to quadratic programming input *formats* rather than the quadratic programming problem itself. For models with an *asymmetric* \mathbf{Q} matrix, either express the quadratic terms algebraically, as described in *Algebraic View* on page 220, or

provide as input $(Q + Q')/2$ instead of Q . This latter approach relies on the identity $Q = (Q + Q')/2 + (Q - Q')/2$ combined with the fact that $(Q - Q')/2$ contributes 0 (zero) to the quadratic objective.

Algebraic View

In the algebraic view, a quadratic objective function is specified as an expressions of the form:

$$c_1x_1 + \dots + c_nx_n + q_{11}x_1x_1 + q_{12}x_1x_2 + \dots + q_{nn}x_nx_n.$$

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 for Entering QPs

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 one-half 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 $x_1 = 1.000000$ and $x_2 = 3.000000$, the result to be expected is 200, **not** 400.

Reformulating QPs to Save Memory

When the **Q** matrix is very dense or extremely large in dimension, excessive memory may be needed to solve the problem as conventionally formulated. However, you may be able to use an alternative formulation to avoid such bottlenecks. Specifically, if you can express **Q** as **FF'**, (where **F** is another matrix, not necessarily square, having fewer nonzeros than **Q**, and **F'** is its transpose) then you can reformulate the QP like this:

```
min c'x + y'y
Ax -b
y - Fx = 0
l <= x <= u
y free
```

In the reformulation, **y** is a vector of free variables, one variable for each column of **F**.

Portfolio optimization models in particular can benefit from this reformulation. In the most common portfolio models, **Q** is a covariance matrix of asset returns, while **F** is the matrix of the deviations of the asset returns from their mean used to compute the covariances. In that reformulation, the number of columns of **F** corresponds to the number of time periods for which returns are measured.

In general, while the number of rows in **F** must match the dimension of the square matrix **Q**, the number of columns of **F** can be fewer. So, even if **Q** is dense and **F** is also dense, you still may reduce the memory requirements to solve the model if **F** has more rows than columns.

Furthermore, if **F** is a sparser matrix than **Q**, this alternative formulation may improve performance even if **F** has more columns than **Q**.

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.

Changing Problem Type in QPs

Concert Technology (that is, applications written in the C++, Java, or .NET API of ILOG CPLEX) 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 usually treated as being of type `qp` if quadratic coefficients are present in the objective function and no quadratic terms are present among the constraints. (Quadratic terms among the constraints may make a problem of type QCP. For more about that type, see *Solving Problems with Quadratic Constraints (QCP)* on page 229.) Otherwise, the problem type is usually `lp`. The issue of problem types that support integer restrictions in conjunction with quadratic variables is discussed in *Solving Mixed Integer Programming Problems (MIP)* on page 245.

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 in the objective function, 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 from the objective function 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 qp term` 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 Q . The quadratic coefficient of an off-diagonal term must be distributed within the Q matrix, and it is always one-half the value of the quadratic algebraic term.

To clarify that terminology, consider this example:

$$\text{Minimize } a + b + 1/2(a^2 + 4ab + 7b^2)$$

subject to $a + b \geq 10$

with these bounds $a \geq 0$ and $b \geq 0$

The off-diagonal quadratic algebraic term in that example is 4, so the quadratic matrix Q is

$$\begin{bmatrix} 1 & 2 \\ 2 & 7 \end{bmatrix}$$

- ◆ In a QP, you can change the quadratic matrix coefficients in the Interactive Optimizer by using the command `change qp term`.
- ◆ 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 Q_{ij} to a value, it also changes 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 qp term
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 $Q = \mathbf{0}$.

When you change quadratic terms, there are still restrictions on the properties of the Q matrix. In a minimization problem, it must be convex, positive semi-definite. In a maximization problem, it must be concave, negative semi-definite. For example, if you change the sense of an objective function in a convex Q matrix from minimization to maximization, you will thus make the problem unsolvable. Likewise, in a convex Q matrix, if you make a 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 can be carried out by the conflict refiner. See *Diagnosing Infeasibility by Refining Conflicts* on page 353.

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 numeric difficulties on LP models (see *Numeric Difficulties* on page 174) 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 311, where automatic transformation with logical constraints can render a problem MIQP.)

The complete program `ilqpex1.cpp` appears online in the standard distribution at [yourCPLEXinstallation/examples/src](#).

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.

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` discussed in ILOG CPLEX *Getting Started*. The Concert forms of this example, `ilqpex1.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 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.

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 discussed in ILOG CPLEX *Getting Started*. The Concert forms 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 online in the standard distribution at [yourCPLEXinstallation/examples/src](#).

Solving Problems with Quadratic Constraints (QCP)

This chapter tells you how to solve *quadratically constrained* programming problems (QCPs), including the special case of second order cone programming (SOCP) problems. This chapter contains sections about:

- ◆ *Identifying a Quadratically Constrained Program (QCP)* on page 229;
- ◆ *Determining Problem Type* on page 233;
- ◆ *Changing Problem Type* on page 239
- ◆ *Changing Quadratic Constraints* on page 240;
- ◆ *Solving with Quadratic Constraints* on page 240;
- ◆ *Numeric Difficulties and Quadratic Constraints* on page 241.

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:

$$\text{Minimize } \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x}$$

subject to $Ax \sim b$

and $a_i^T x + x^T Q_i x \leq r_i$ for $i=1, \dots, q$

with these bounds $l \leq x \leq u$

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 Q_i matrix be positive semi-definite (PSD) or that the constraint must be in the form of a second order cone. The following sections offer more information about these concepts.

Convexity

The inequality $x^2 + y^2 \leq 1$ 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.

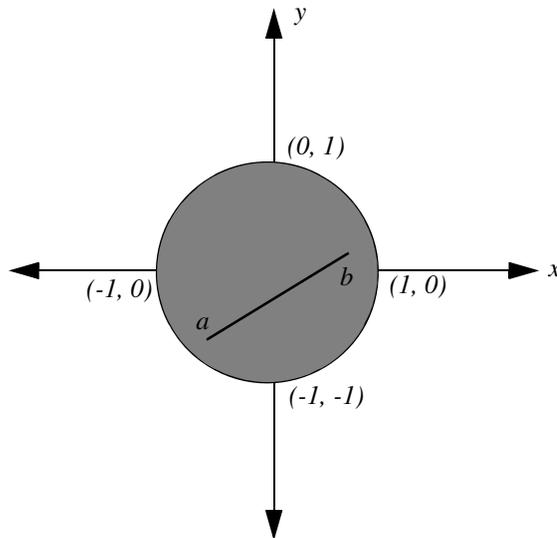


Figure 12.1 $x^2 + y^2 \leq 1$ is convex

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.

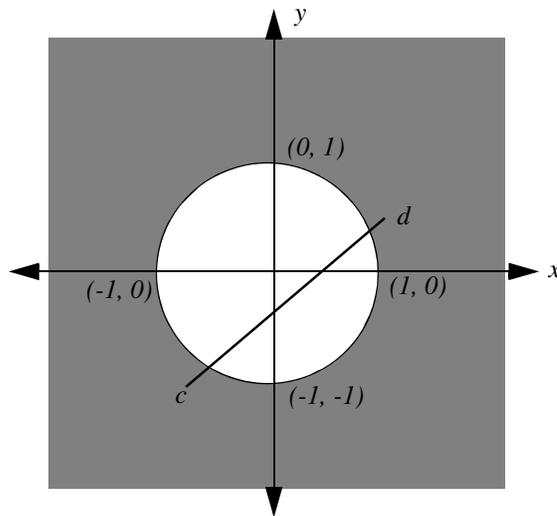


Figure 12.2 $x^2 + y^2 \geq 1$ is not convex

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.

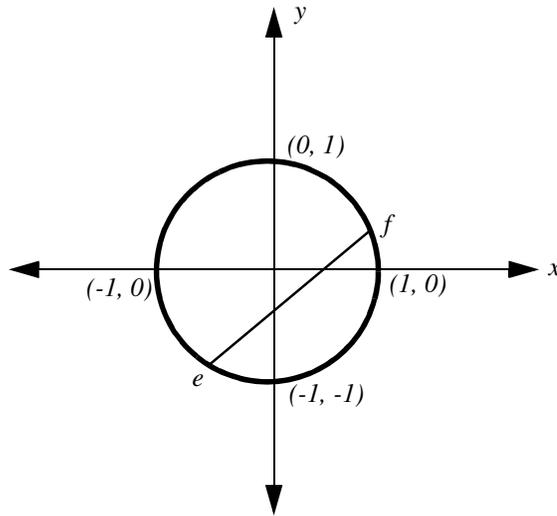


Figure 12.3 $x^2 + y^2 = 1$ is not convex

Semi-definiteness

Identifying a Quadratically Constrained Program (QCP) on page 229 explained that the quadratic matrix in each constraint must be positive semi-definite (PSD), thus providing convexity. A matrix \mathbf{Q}_i is PSD if $\mathbf{x}^T \mathbf{Q}_i \mathbf{x} \geq 0$ for every vector \mathbf{x} , whether or not \mathbf{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 218.

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.

Second Order Cone Programming (SOCP)

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 229. 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 second-order cone constraint:

$$-c_0 x_0^2 + \sum c_i x_i^2 \leq 0$$

That formulation is distinguished primarily by the specific signs of the coefficients c and by the lack of a linear term, where x_0 is a nonnegative variable

Determining Problem Type

ILOG CPLEX determines the type of your QCP model according to various criteria. The following sections explain more about problem type.

- ◆ *Concert Technology and QCP Problem Type* on page 233
- ◆ *Callable Library and QCP Problem Type* on page 233
- ◆ *Interactive Optimizer and QCP Problem Type* on page 233
- ◆ *File Formats and QCP Problem Type* on page 233

Concert Technology and QCP 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.

Callable Library and QCP Problem Type

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 222.

Interactive Optimizer and QCP Problem Type

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.

File Formats and QCP Problem Type

ILOG CPLEX supports the definition of quadratic constraints in SAV files with the `.sav` file extension, in LP files with the `.lp` file extension, and in MPS files with the `.mps` file extension. In LP files, you can state your quadratic constraints in the **subject to** section of the file. For more detail about representing QPC models in MPS file format, see the *ILOG CPLEX File Format Reference Manual*, especially the section *Quadratically Constrained*

Programs (QCP) in MPS Files on page 27. Here is a sample of a file including quadratic constraints in MPS format.

```
NAME          /ilog/models/miqcp/all/p0033_qc1.lp.gz
ROWS
N  R100
L  R118
L  R119
L  R120
L  R121
L  R122
L  R123
L  R124
L  R125
L  R126
L  R127
L  R128
L  ZBESTROW
L  QC1
L  QC2
L  QC3
L  QC4
```

COLUMNS		
MARK0000	'MARKER'	'INTORG'
C157	R100	171
C157	R122	-300
C157	R123	-300
C158	R100	171
C158	R126	-300
C158	R127	-300
C159	R100	171
C159	R119	300
C159	R120	-300
C159	R123	-300
C159	QC1	1
C160	R100	171
C160	R119	300
C160	R120	-300
C160	R121	-300
C161	R100	163
C161	R119	285
C161	R120	-285
C161	R124	-285
C161	R125	-285
C162	R100	162
C162	R119	285
C162	R120	-285
C162	R122	-285
C162	R123	-285
C163	R100	163
C163	R128	-285
C164	R100	69
C164	R119	265
C164	R120	-265
C164	R124	-265
C164	R125	-265
C165	R100	69
C165	R119	265
C165	R120	-265
C165	R122	-265
C165	R123	-265
C166	R100	183
C166	R118	-230

C167	R100	183
C167	R124	-230
C167	R125	-230
C168	R100	183
C168	R119	230
C168	R120	-230
C168	R125	-230
C169	R100	183
C169	R119	230
C169	R120	-230
C169	R123	-230
C170	R100	49
C170	R119	190
C170	R120	-190
C170	R122	-190
C170	R123	-190
C171	R100	183
C172	R100	258
C172	R118	-200
C173	R100	517
C173	R118	-400
C174	R100	250
C174	R126	-200
C174	R127	-200
C175	R100	500
C175	R126	-400
C175	R127	-400
C176	R100	250
C176	R127	-200
C177	R100	500
C177	R127	-400
C178	R100	159
C178	R119	200
C178	R120	-200
C178	R124	-200
C178	R125	-200
C179	R100	318
C179	R119	400
C179	R120	-400
C179	R124	-400
C179	R125	-400

C180	R100	159
C180	R119	200
C180	R120	-200
C180	R125	-200
C181	R100	318
C181	R119	400
C181	R120	-400
C181	R125	-400
C182	R100	159
C182	R119	200
C182	R120	-200
C182	R122	-200
C182	R123	-200
C183	R100	318
C183	R119	400
C183	R120	-400
C183	R122	-400
C183	R123	-400
C184	R100	159
C184	R119	200
C184	R120	-200
C184	R123	-200
C185	R100	318
C185	R119	400
C185	R120	-400
C185	R123	-400
C186	R100	114
C186	R119	200
C186	R120	-200
C186	R121	-200
C187	R100	228
C187	R119	400
C187	R120	-400
C187	R121	-400
C188	R100	159
C188	R128	-200
C189	R100	318
C189	R128	-400
MARK0001	'MARKER'	'INTEND'

RHS		
rhs	R118	-5
rhs	R119	2700
rhs	R120	-2600
rhs	R121	-100
rhs	R122	-900
rhs	R123	-1656
rhs	R124	-335
rhs	R125	-1026
rhs	R126	-5
rhs	R127	-500
rhs	R128	-270
rhs	QC1	1
rhs	QC2	2
rhs	QC3	1
rhs	QC4	1
BOUNDS		
UP bnd	C157	1
UP bnd	C158	1
UP bnd	C159	1
UP bnd	C160	1
UP bnd	C161	1
UP bnd	C162	1
UP bnd	C163	1
UP bnd	C164	1
UP bnd	C165	1
UP bnd	C166	1
UP bnd	C167	1
UP bnd	C168	1
UP bnd	C169	1
UP bnd	C170	1
UP bnd	C171	1
UP bnd	C172	1
UP bnd	C173	1
UP bnd	C174	1
UP bnd	C175	1
UP bnd	C176	1
UP bnd	C177	1
UP bnd	C178	1
UP bnd	C179	1
UP bnd	C180	1
UP bnd	C181	1
UP bnd	C182	1
UP bnd	C183	1
UP bnd	C184	1
UP bnd	C185	1
UP bnd	C186	1
UP bnd	C187	1
UP bnd	C188	1
UP bnd	C189	1

```

QMATRIX
  C158      C158      1
  C158      C189      0.5
  C189      C158      0.5
  C189      C189      1
QCMATRIX  QC1
  C157      C157      1
  C157      C158      0.5
  C158      C157      0.5
  C158      C158      1
  C159      C159      1
  C160      C160      1
QCMATRIX  QC2
  C161      C161      2
  C162      C162      2
  C163      C163      1
QCMATRIX  QC3
  C164      C164      1
  C165      C165      1
QCMATRIX  QC4
  C166      C166      1
  C167      C167      1
  C168      C168      1
  C169      C169      1
  C171      C171      1
ENDATA

```

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 76. To add constraints to a model in the .NET framework, see *ILOG Concert Technology for .NET Users* on page 95.

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 59 and about Java applications in *Modifying the Model* on page 92.

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 239. 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 239.

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.

- ◆ 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.

Numeric Difficulties and Quadratic Constraints

A word of warning: numeric difficulties are likely to be more acute for QCP than for LP or QP. Symptoms include:

- ◆ lack of convergence to an optimal solution;
- ◆ violation of constraints.

Consequently, you will need to scale your variables carefully so that units of measure are roughly comparable among them.

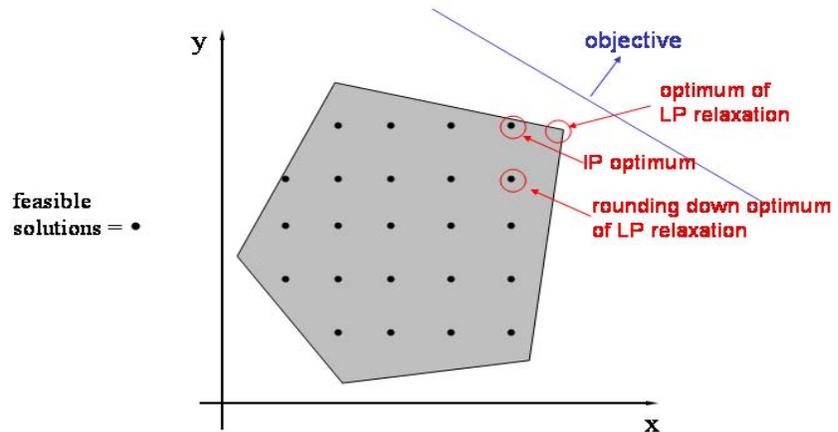
Examples: QCP

For examples of QCPs, see these variations of the same problem in *yourCPLEXhome/examples/src*:

- ◆ `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 245
- ◆ *Using Special Ordered Sets (SOS)* on page 291
- ◆ *Using Semi-Continuous Variables: a Rates Example* on page 295
- ◆ *Using Piecewise Linear Functions in Optimization: a Transport Example* on page 299
- ◆ *Logical Constraints in Optimization* on page 311
- ◆ *Using Indicator Constraints* on page 317
- ◆ *Using Logical Constraints: Food Manufacture 2* on page 321
- ◆ *Early Tardy Scheduling* on page 329
- ◆ *Using Column Generation: a Cutting Stock Example* on page 335

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 246
- ◆ *Considering Preliminary Issues* on page 247
- ◆ *Using the Mixed Integer Optimizer* on page 251
- ◆ *Tuning Performance Features of the Mixed Integer Optimizer* on page 254
- ◆ *Using the MIP Solution* on page 271
- ◆ *Progress Reports: Interpreting the Node Log* on page 273
- ◆ *Troubleshooting MIP Performance Problems* on page 277
- ◆ *Examples: Optimizing a Basic MIP Problem* on page 288
- ◆ *Example: Reading a MIP Problem from a File* on page 289

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 the MILP, MIQP, or MIQCP 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: a Rates Example* on page 295 says a bit more about semi-continuous variables later in this manual. Special Ordered Sets (SOS) are discussed in *Using Special Ordered Sets (SOS)* on page 291. 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 formulation illustrates a mixed integer programming problem, which is solved in the example program `ilomipex1.cpp / mipex1.c`, discussed later in this chapter:

$$\begin{array}{ll} \text{Maximize} & x_1 + 2x_2 + 3x_3 + x_4 \\ \text{subject to} & - x_1 + x_2 + x_3 + 10x_4 \leq 20 \\ & x_1 - 3x_2 + x_3 \leq 30 \\ & x_2 - 3.5x_4 = 0 \\ \text{with these bounds} & 0 \leq x_1 \leq 40 \\ & 0 \leq x_2 \leq +\infty \\ & 0 \leq x_3 \leq +\infty \\ & 2 \leq x_4 \leq 3 \\ & x_4 \text{ integer} \end{array}$$

Considering Preliminary Issues

When you are optimizing a MIP, there are a few preliminary issues that you need to consider to get the most out of ILOG CPLEX. The following sections cover such topics as entering variable type, displaying MIPs in the Interactive Optimizer, determining the problem type, and switching to the fixed form of your problem.

- ◆ *Entering MIP Problems* on page 247
- ◆ *Displaying MIP Problems* on page 248
- ◆ *Changing Problem Type in MIPs* on page 249
- ◆ *Changing Variable Type* on page 250

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	.NET API
binary	<code>IloNumVar::Type::ILOBOOL</code>	<code>IloNumVarType.Bool</code>	<code>NumVarType.Bool</code>
integer	<code>IloNumVar::Type::ILOINT</code>	<code>IloNumVarType.Int</code>	<code>NumVarType.Int</code>
semi-continuous	<code>IloSemiContVar::Type::ILONUM</code>	<code>IloNumVarType.Float</code>	<code>NumVarType.Float</code>
semi-integer	<code>IloSemiContVar::Type::ILOINT</code>	<code>IloNumVarType.Int</code>	<code>NumVarType.Int</code>

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 246, you type this:

```
generals
x4
```

You may also read MIP data from a formatted file, just as you do for linear programming problems. *Understanding File Formats* on page 142 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
<code>display problem binaries</code>	lists variables restricted to binary values
<code>display problem generals</code>	lists variables restricted to integer values
<code>display problem semi-continuous</code>	lists variables of type semi-continuous and semi-integer
<code>display problem integers</code>	lists all of the above
<code>display problem sos</code>	lists the names of variables in one or more Special Ordered Sets
<code>display problem stats</code>	lists LP statistics plus: <ul style="list-style-type: none"> • 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 `CPXgetcctype` 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 declaration 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 milpf, miqp, or miqcp. Conversely, if you have entered an MILP, MIQP, or MIQCP 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 continuous 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`, `miqp`, or `miqcp`

indicating that you want ILOG CPLEX to treat the problem as an MILP, MIQP, or MIQCP, 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_MIQP`, or `CPXPROB_MIQCP` for the MILP, MIQP, and MIQCP 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_QP`, or `CPXPROB_QCP`.

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 form 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).

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 numeric 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, use one of these approaches:

- ◆ In the Interactive Optimizer, use the `mipopt` command.
- ◆ In Concert Technology, with the `IloCplex` method `solve`.
- ◆ In the Callable Library, use the `CPXmipopt` routine.

Emphasizing Feasibility and Optimality

The following section, *Tuning 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; it balances optimality and integer feasibility.

At this default `MIPEmphasis` setting of 0 (that is, `MIPEmphasisBalanced` in Concert Technology or `CPX_MIPEMPHASIS_BALANCED` in the Callable Library), 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` in Concert Technology 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` in Concert Technology 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.

The fourth choice for `MIPEmphasis`, 3 (`MIPEmphasisBestBound` in Concert Technology 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 so far: 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 with emphasis 3, but the user has the option if that is what is wanted.

Terminating MIP Optimization

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 of the branch & cut tree have been processed. Optimality in this case is relative to whatever tolerances and optimality criteria you have set. For example, ILOG CPLEX considers any user-supplied cutoff value (such as `CutLo` or `CutUp`) as well as the objective difference parameter (`ObjDiff`) when it treats nodes during branch & cut. Thus these settings indirectly affect termination.

An important termination criterion that the user can set explicitly is the MIP gap tolerance. In fact, there are two such tolerances: a relative MIP gap tolerance that is commonly used, and an absolute MIP gap tolerance that is appropriate in cases where the expected optimal objective function is quite small in magnitude. The default value of the relative MIP gap tolerance is 1e-4; the default value of the absolute MIP gap tolerance is 1e-6. These default values indicate to CPLEX to stop when an integer feasible solution has been proved to be within 0.01% of optimality. On a difficult model with input data obtained with only

approximate accuracy, where a proved optimum is thought to be unlikely within a reasonable amount of computation time, a user might choose a larger relative MIP Gap, for example, 0.05 (corresponding to 5%) to allow early termination. Conversely, in a model where the objective function amounts to billions of dollars and the data are accurate to a degree that further processing is worthwhile, a tighter relative MIP Gap (even 0.0) may be advantageous to avoid any chance of missing the best possible solution.

ILOG CPLEX also 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 13.4 summarizes those parameters and their purpose.

Table 13.4 *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
number of integer solutions	IntSolLim	CPX_PARAM_INTSOLLIM	mip limits solutions
relative MIP gap tolerance	EpGap	CPX_PARAM_EPGAP	mip tolerances mipgap
absolute MIP gap tolerance	EpAGap	CPX_PARAM_EPAGAP	mip tolerances absmipgap

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. For suggestions about overcoming such errors, see *Troubleshooting MIP Performance Problems* on page 277.

Tuning 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 features and parameters that are the most likely to offer help on such models.

- ◆ *Branch & Cut* on page 255
- ◆ *Probing* on page 260

- ◆ *Cuts* on page 261
 - ◆ *Heuristics* on page 265
 - ◆ *Preprocessing: Presolver and Aggregator* on page 267
 - ◆ *Starting from a Solution* on page 269
 - ◆ *Issuing Priority Orders* on page 270
-

Branch & Cut

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.

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 261 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.

- ◆ *Applying Cutoff Values* on page 256
- ◆ *Applying Tolerance Parameters* on page 256
- ◆ *Applying Heuristics* on page 256
- ◆ *When an Integer Solution Is Found: the Incumbent* on page 256
- ◆ *Controlling Strategies: Diving and Backtracking* on page 257
- ◆ *Selecting Nodes* on page 258
- ◆ *Selecting Variables* on page 259
- ◆ *Changing Branching Direction* on page 259
- ◆ *Solving Subproblems* on page 260
- ◆ *Using Node Files* on page 260

Applying Cutoff Values

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.

Applying Tolerance Parameters

ILOG CPLEX will use the value of the best integer solution found so far, as modified by the tolerance parameters `ObjDif` (absolute objective function difference) or `RelObjDif` (relative objective function difference) 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.

Applying Heuristics

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.

When an Integer Solution Is Found: the Incumbent

After 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.

Controlling Strategies: Diving and Backtracking

You control the path that CPLEX traverses in the tree through several parameters, as summarized in Table 13.5.

Table 13.5 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, BBInterval, i)
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.

Selecting Nodes

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 13.6 *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 use an estimate of 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.

Selecting Variables

After a node has been selected, the variable selection parameter, `VarSel`, influences which variable is chosen for branching at that node.

Table 13.7 *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 automatically determines 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 form of pseudo costs.

Changing Branching Direction

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 13.8 *BrDir Parameter Settings for Branching Direction Choice*

BrDir Setting	Symbolic Value	Branching Direction Choice
-1	CPX_BRANCH_DOWN	Branch downward
0 (Default)	CPX_BRANCH_GLOBAL	ILOG CPLEX automatically determines each branch direction.
1	CPX_BRANCH_UP	Branch upward

Priority orders complement the behavior of these parameters. They are introduced in *Issuing Priority Orders* on page 270. 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.

Solving Subproblems

ILOG CPLEX allows you to distinguish the algorithm applied to the initial relaxation of your problem from the algorithm applied to other continuous subproblems of a MIP. This distinction between initial relaxation and the other MIP subproblems may be useful when you have special information about the nature of your model. In this context, "other MIP subproblems" includes nodes of the branch & cut tree, problems re-solved after cutting plane passes, problems solved by node heuristics, and so forth.

The parameter `RootAlg` (`CPX_PARAM_STARTALG`) enables you to specify which algorithm for ILOG CPLEX to apply to the initial relaxation.

The parameter `NodeAlg` (`CPX_PARAM_SUBALG`) lets you specify the algorithm applied to other continuous subproblems.

For more detail about these parameters, see *Unsatisfactory Subproblem Optimization* on page 286.

Using Node Files

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 can be an effective technique which uses disk space to augment RAM, at little or no penalty in terms of solution speed.

The node-file storage-feature enables you to store some parts of the branch & cut tree in files while the branch & cut algorithm is being applied. 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 as managed by ILOG CPLEX itself offers a much better option in terms of memory use and performance time than relying on swap space as managed by your operating system in this context.

For more about the parameters controlling node files, see *Use Node Files for Storage* on page 282.

Probing

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 (zero) or 1 (one). 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 default setting of the `Probe` parameter (0 (zero)), ILOG CPLEX will automatically determine an 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 an increasing level of 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`.

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* on page 262
- ◆ *Cover Cuts* on page 262
- ◆ *Disjunctive Cuts* on page 262
- ◆ *Flow Cover Cuts* on page 262
- ◆ *Flow Path Cuts* on page 262
- ◆ *Gomory Fractional Cuts* on page 262
- ◆ *Generalized Upper Bound (GUB) Cover Cuts* on page 263
- ◆ *Implied Bound Cuts* on page 263
- ◆ *Mixed Integer Rounding (MIR) Cuts* on page 263
- ◆ *Adding Cuts and Re-Optimizing* on page 263
- ◆ *Counting Cuts* on page 263

◆ *Parameters Affecting Cuts* on page 264

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 righthand 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 righthand 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.

Counting Cuts

To know the number of cuts added to a problem during MIP optimization, implement one of the following techniques in your application:

- ◆ For Concert Technology users:

Declare an output stream in the conventional C++, Java, or .NET way. Capture the output and examine it for a string identifying the type of cut you want to count. For example, look for the string "Cover cuts applied:" to know the number of cover cuts, or look for the string "Cover cuts applied:" to know the number of clique cuts. The number of cuts of that type follows immediately after the string.

- ◆ For Callable Library users:

Before calling the MIP optimizer in your application, use the routine `CPXsetmipcallback` to set up a MIP callback function that queries the cut count. Use the routine `CPXgetcallbackinfo` to count the number of cuts of each type of interest to you.

Parameters Affecting Cuts

Parameters control the way each class of cuts is used. Those parameters are listed in Table 13.9.

Table 13.9 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 clique cuts, cover cuts, and disjunctive cuts, a setting of 3 is permitted, which indicates that the specified type of cut should be generated very aggressively.

In the Interactive Optimizer, the command `set mip cuts all i` applies the value `i` to all types 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 means that no cuts will be generated; this setting 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 nondefault 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 lacking an indication of whether it has found the best solution possible.

ILOG CPLEX provides these families of heuristics to find integer solutions at nodes (including the root node) during the branch & cut procedure:

- ◆ *Node Heuristic* on page 265
- ◆ *Relaxation Induced Neighborhood Search (RINS) Heuristic* on page 265
- ◆ *Solution Polishing* on page 266

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.

Solution Polishing

Solution polishing can be used to improve the best known solution at the end of branch & cut if optimality has not been proven. Alternatively, it can be used instead of the branch & cut algorithm if an initial solution can be found at the root node. If Solution Polishing is used as an alternative algorithm to branch & cut, optimality may not be proven, even if the optimal solution is found.

A parameter enables you to specify the amount of time ILOG CPLEX spends polishing the best solution found. The default value of this parameter is 0.0, so that by default no separate polishing phase is performed. The parameter accepts any nonnegative value, to set a limit in seconds.

- `PolishTime` in Concert Technology
- `CPX_PARAM_POLISHTIME` in the Callable Library
- `mip limit polishtime` in the Interactive Optimizer

If a MIP optimization has located a feasible solution and already terminated, you can invoke polishing alone in the same application call or interactive session by following these steps:

1. Set the polishing time parameter to a positive value.
2. Set the ordinary time limit to 0.0.
 - `TiLim` in Concert Technology
 - `CPX_PARAM_TILIM` in the Callable Library
 - `timelimit` in the Interactive Optimizer
3. Call the optimizer again.

Remember to leave the advanced indicator parameter at its default value of 1 (2 is also acceptable) so that the polishing will proceed from the advanced start.

- `AdvInd` in Concert Technology

- CPX_PARAM_ADVIND in the Callable Library
- advance in the Interactive Optimizer

As with the RINS heuristic, the subMIP node-limit parameter also controls aspects of the work that solution polishing performs.

- SubMIPNodeLim in Concert Technology
- CPX_PARAM_SUBMIPNODELIM in the Callable Library
- mip limits submipnodelim in the Interactive Optimizer

Solution polishing always requires the presence of a feasible solution from which to start.

Solution polishing is much more time-intensive than any of the other heuristics, but can yield better solutions in some situations.

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 13.10 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 13.10 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 (bool)	CPX_PARAM_PERIND (int)	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
set preprocessing reresolve	RepeatPresolve	CPX_PARAM_REPEATPRESOLVE	automatic by default

These and other parameters also control the behavior of preprocessing of the continuous subproblem (LP, QP, or QCP) solved during a MIP optimization. See *Preprocessing* on

page 166 for further details about these parameters in that context. The following discussion pertains to these parameters specifically in MIP preprocessing.

While preprocessing, ILOG CPLEX 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 attempts to reduce coefficients of constraints 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, possibly 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.

The `RelaxPreInd` parameter controls whether an additional round of presolve is applied before ILOG CPLEX solves the continuous subproblem at the root relaxation. Often the root relaxation is the single most time-consuming subproblem solved during branch-and-cut. Certain additional presolve reductions are possible when MIP restrictions are **not** present, and on difficult models this extra step will often pay off in faster root-solve times. Even when there is no appreciable benefit, there is usually no harm either. However, the `RelaxPreInd` parameter is available if you want to explore whether skipping the additional presolve step will improve overall solution speed, for example, if you are solving a long sequence of very easy models and need maximum speed on each one.

It is possible to apply preprocessing a second time, after cuts and other analyses have been performed and before branching begins. If your models tend to require a lot of branching, this technique is sometimes useful in further tightening the formulation. Use the `RepeatPresolve` parameter (`CPX_PARAM_REPEATPRESOLVE` in the Callable Library) to invoke this additional step. Its default value of -1 means that ILOG CPLEX makes the decision internally whether to repeat presolve; 0 (zero) turns off this feature unconditionally, while positive values allow you control over which aspects of the preprocessed model are re-examined during preprocessing and whether additional cuts are then permitted. Table 13.11 summarizes the possible values of this parameter.

Table 13.11 Values of RepeatPresolve Parameter

Value	Effect
-1	Automatic (default)
0	Turn off repeat presolve
1	Repeat presolve without cuts
2	Repeat presolve with cuts
3	Repeat presolve with cuts and allow new root cuts

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. When you provide such a starting solution, you may invoke relaxation induced neighborhood search (its RINS heuristic) or solution polishing to improve the given solution. This first integer solution may include continuous and discrete variables of various types, such as semi-continuous variables or those in lazy constraints, linearized constraints, or special ordered sets.

If you specify values for all discrete variables, ILOG CPLEX will check the validity of the values as an integer-feasible solution; if you specify values for only a portion of the discrete variables, CPLEX will attempt to fill in the missing values in a way that leads to an integer-feasible solution. If the specified values do not lead directly to an integer-feasible solution, CPLEX will apply a quick heuristic to try to repair the MIP Start. The number of times that CPLEX applies the heuristic is controlled by the repair tries parameter (`RepairTries` in Concert Technology, `CPX_PARAM_REPAIRTRIES` in the Callable Library). If this process succeeds, the solution will be treated as an integer solution of the current problem.

After a MIP start has been established for your model, its use is controlled by the advanced indicator parameter (`AdvInd` in Concert Technology; `CPX_PARAM_ADVIND` in the Callable Library). At its default setting of 1, the MIP start values that you specify are used. If you set `AdvInd` to the value 0 (zero), then the MIP Start will **not** be used. The other optional setting for `AdvInd`, 2, has meaning only for continuous models being solved by one of the simplex optimizers; for MIP, the setting 2 has the same effect as 1 (one).

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 application, 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.

Issuing 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 13.12.

Table 13.12 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.

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 `CPXgetx`.

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
```

Further solution information, such as the optimal values of the slack variables for the constraints, can be written to a file in the SOL format. See the description of this file format in the *ILOG CPLEX File Formats Reference Manual* in *SOL File Format: Solution Files* on page 38.

For any of the MIP problem types, the following additional solution information is available in the Interactive Optimizer through the `display` command after optimization has produced a solution:

- 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 form 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.

If you wish to access dual information in such a problem, first optimize your MILP problem to create the **fixed** MILP problem; then re-optimize it, like this:

- ◆ In Concert Technology, call the method `solveFixed`. (There is no explicit problem type in Concert Technology, so there is no need to change the problem type as in other components.)
- ◆ In the Callable Library, call the routine `CPXchgprobtype` with the argument `CPXPROB_FIXEDMILP` as the problem type and then call `CPXlpopt`.
- ◆ In the Interactive Optimizer, use these commands to change the problem type and re-optimize:
 - `change problem fixed_milp`
 - `optimize`

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 13.13 summarizes its possible values and their effects.

Table 13.13 *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 $\text{abs}(\text{best integer} - \text{best node}) / (1e-10 + \text{abs}(\text{best integer}))$.

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 246 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;
- N means that cuts added to the node LP resulted in an integer feasible solution; no branching was required.
- ◆ 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 37 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 37 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.

Tuning Performance Features of the Mixed Integer Optimizer on page 254 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 278
- ◆ *Trouble Finding More than One Feasible Solution* on page 278
- ◆ *Large Number of Unhelpful Cuts* on page 279
- ◆ *Lack of Movement in the Best Node* on page 279
- ◆ *Time Wasted on Overly Tight Optimality Criteria* on page 279
- ◆ *Slightly Infeasible Integer Variables* on page 280
- ◆ *Running out of Memory* on page 281
- ◆ *Difficulty Solving Subproblems: Overcoming Degeneracy* on page 285

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 *Solving LPs: Simplex Optimizers* on page 161, *Solving Problems with a Quadratic Objective (QP)* on page 217, and *Solving Problems with Quadratic Constraints (QCP)* on page 229 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` for example to turn off only the cover cuts) 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 13.9 on page 264.

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, E_{pRHS} , from its default value of $1e-6$ to as low as $1e-9$, its minimum, and also tighten E_{pInt} 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 E_{pInt} all the way to zero, preferably in conjunction with a tightened E_{pRHS} setting. This very tight integrality tolerance directs CPLEX to attempt to branch on any integer infeasibility, no matter how small. Numeric 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 in many others. In cases where the integrality violation even after branching remains above E_{pInt} or is above $1e-10$ when E_{pInt} has been set to a value smaller than that, a solution status returned will be `CPX_STAT_OPTIMAL_INFEAS` instead of the usual `CPX_STAT_OPTIMAL`. In most cases a solution with status `CPX_STAT_OPTIMAL_INFEAS` will be satisfactory, and reflects only roundoff error after presolve uncrush, but extra care in using the solution may be advisable in numerically sensitive formulations.

Running out of Memory

A very common difficulty with MIPs is running out of memory. This problem almost always occurs when the branch & cut tree becomes so large that insufficient memory remains to solve a continuous LP, QP, or QCP subproblem. (In the rare case that the dimensions of a very large model are themselves the main contributor to memory consumption, you can try adjusting the memory emphasis parameter, as described in *Lack of Memory* on page 173.) 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 at 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. Since generation of cutting planes is not a frequent cause of memory consumption, apply these recommendations only as a last resort, after trying to resolve the problem with less drastic measures.

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 282, for other choices of what should happen when `WorkMem` is exceeded. That topic explains how to indicate to CPLEX that it should use disk for working storage.

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. At default settings, the limit is infinity ($1e^{+75}$), but you can set it to a lower value (in megabytes).

Use Node Files for Storage

As noted in *Using Node Files* on page 260, ILOG CPLEX offers a node-file storage-feature to store some parts of the branch & cut tree in files as it progresses through its search. This feature allows ILOG CPLEX to explore more nodes within a smaller amount of computer memory. It also includes several options to reduce the use of physical memory. Importantly, it entails only a very small increase in runtime. In terms of performance, node-file storage offers a much better option than relying on generic swap space managed by your operating system.

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 the following sample output.

```
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)
```

There are several parameters that control the use of node files. They are:

- ◆ `WorkMem` in Concert Technology or `CPX_PARAM_WORKMEM` in the Callable Library
- ◆ `NodeFileInd` in Concert Technology or `CPX_PARAM_NODEFILEIND` in the Callable Library
- ◆ `TreLim` in Concert Technology or `CPX_PARAM_TRELIM` in the Callable Library
- ◆ `WorkDir` in Concert Technology or `CPX_PARAM_WORKDIR` in the Callable Library

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. The default value of the `WorkMem` parameter is 128 megabytes. Setting it to higher values, even on a machine with very large memory, can be expected to result in only marginally improved efficiency. However, it is advisable to reduce this setting to approximately half the available memory of your machine if your machine has less than 256 megabytes of RAM to avoid defeating the purpose of node files, a situation that would occur if your application inadvertently triggers the swap space of your operating system.

When tree storage size exceeds the limit defined by `WorkMem`, and if the tree-memory limit has not been exceeded, then what happens next is determined by the setting of the node file indicator (`NodeFileInd` in Concert Technology or `CPX_PARAM_NODEFILEIND` in the Callable Library). If that parameter is set to zero, then optimization proceeds with the tree stored in memory until ILOG CPLEX reaches the tree memory limit (`TreLim` in Concert Technology or `CPX_PARAM_TRELIM` in the Callable Library). If the node file indicator 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). Table 13.14 summarizes these different options.

Table 13.14 *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

Among the memory conservation tactics employed by ILOG CPLEX when the memory emphasis parameter has been set, the maximum setting for the node file indicator is automatically chosen, so that node-file storage will go to disk. You may still wish to adjust the working memory or tree limit parameters to fit the capabilities of your computer.

In cases where node files are written to disk, ILOG CPLEX will create a temporary subdirectory under the directory specified by the working directory parameter (`WorkDir` in Concert Technology or `CPX_PARAM_WORKDIR` in the Callable Library). 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.

Node files could grow very large. Use the parameter `TreLim` (`CPX_PARAM_TRELIM`) to limit the size of the tree so that it does not exceed available disk space, when you choose `NodeFileInd` (`CPX_PARAM_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 13.6 on page 258. 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 for variable selection; that is, the parameter `VarSel` (`CPX_PARAM_VARSSEL`) set to the value 3. 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: Overcoming Degeneracy

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 *Unsatisfactory Subproblem Optimization* on page 286. A stronger dual pricing algorithm, such as dual steepest-edge pricing (the parameter `DPriInd` or `CPX_PARAM_DPRIIND` set to 2), could also be considered.

If the subproblems are dual degenerate, then consider using the primal simplex optimizer for the subproblems. You make this change by setting the parameter `NodeAlg` (`CPX_PARAM_SUBALG`) to 1 (one).

Unsatisfactory Subproblem Optimization

In some problems, you can improve performance by evaluating how the continuous 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 13.15 summarizes the values available for the `RootAlg` parameter.

To set this parameter:

- ◆ In Concert Technology, use the `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.
- ◆ In the Interactive Optimizer, use the command `set mip strategy startalgorithm` with the value to indicate the optimizer you want.

Table 13.15 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 13.15.

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).

Examples: Optimizing a Basic MIP Problem

These examples illustrate how to optimize a MIP with the ILOG CPLEX Component Libraries:

- ◆ *ilomipex1.cpp* on page 288 for the C++ API of Concert Technology;
- ◆ *mipex1.c* on page 288 for the C API of the Callable Library.

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 246.
- ◆ The routine `populatebyrow` added the variables, objective, and constraints to the model created by the method `IloModel model(env)`.

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 246.
- ◆ 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`, `CPXgetobjval`, `CPXgetx`, and `CPXgetslack` (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 `CPXchgproctype` 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.

Example: Reading a MIP Problem from a File

These examples show you how to solve a MIP with the Component Libraries when the problem data is stored in a file:

- ◆ *ilomipex2.cpp* on page 289 for the C++ API of Concert Technology;
- ◆ *mipex2.c* on page 289 for the C API of the Callable Library.

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.

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 does not 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.

Using Special Ordered Sets (SOS)

ILOG CPLEX enables you to define special ordered sets (SOSs) in the model of your problem as a way to specify integrality conditions. The following sections tell you more about special ordered sets.

- ◆ *What Is a Special Ordered Set (SOS)?* on page 291
- ◆ *Example: SOS Type 1 for Sizing a Warehouse* on page 292
- ◆ *Declaring SOS Members* on page 293
- ◆ *Examples: Using SOS and Priority* on page 293

What Is a Special Ordered Set (SOS)?

A special ordered set (SOS) is an additional way to specify integrality conditions in a model. In particular, a special ordered set is a way to restrict the number of nonzero solution values among a specified set of variables in a model. There are various types of SOS:

- ◆ SOS Type 1 is a set of variables where at most one variable may be nonzero.
- ◆ SOS Type 2 is a set of variables where at most two variables may be nonzero. If two variables are nonzero, they must be adjacent in the set.

The members of a special ordered set (SOS) individually may be continuous or discrete variables in any combination. However, even when all the members are themselves

continuous, a model containing one or more special ordered sets (SOSs) becomes a discrete optimization problem requiring the mixed integer optimizer for its solution.

ILOG CPLEX uses special branching strategies to take advantage of SOSs. For many classes of problems, these branching strategies can significantly improve performance. These 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 ordered relationship among the variables (such that weights cannot be specified or would not be meaningful), other formulations should be used instead of a special ordered set.

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 features of Concert Technology.
 - In the C++ API, use the classes `IloSOS1`, `IloSOS2`, or the methods `IloCplex::addSOS1` or `addSOS2`.
 - In the Java API, use the interfaces `IloSOS1` or `IloSOS2`, or use the methods `IloCplex.addSOS1` or `addSOS2`.
 - In the .NET API, use the interfaces `ISOS1` or `ISOS2`, or use the methods `CplexModeler.AddSOS1` or `CplexModeler.AddSOS2`.
- ◆ Use routines from the Callable Library, such as `CPXaddsos` or `CPXcopysos`.
- ◆ Use SOS declarations within an LP file (that is, one in LP format with the file extension `.lp`). Conventions for declaring SOS information in LP files are documented in the *ILOG CPLEX File Format Reference Manual*.
- ◆ Use SOS declarations within an MPS file (that is, one in MPS format with the file extension `.mps`). If you already have MPS files with SOS information, you may prefer this option, but keep in mind that this way of declaring an SOS supports the fewest number of digits of precision in the data. Conventions for declaring SOS information in MPS files are documented in the *ILOG CPLEX File Format Reference Manual*.

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*.) Each of those ways of declaring SOS members allows you to specify weights.

The SOS example, *Example: SOS Type 1 for Sizing a Warehouse* on page 292, used the coefficients of the warehouse capacity constraint to assign weights.

Examples: Using SOS and Priority

These examples illustrate how to use SOS and priority orders:

- ◆ *ilomipex3.cpp* on page 293 for the C++ API of Concert Technology;
- ◆ *mipex3.c* on page 294 for the C API of the Callable Library.

ilomipex3.cpp

This example 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 priority order.
-

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 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 priority order to files by calling `CPXordwrite`.
- ◆ The routine `CPXwriteprob` writes the problem with the constraints and SOSs to disk before the example copies the SOS and priority order to verify that the base problem was copied correctly.

Using Semi-Continuous Variables: a Rates Example

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:

- ◆ *What Are Semi-Continuous Variables?* on page 296
- ◆ *Describing the Problem* on page 296
- ◆ *Representing the Problem* on page 297
- ◆ *Building a Model* on page 297
- ◆ *Solving the Problem* on page 298
- ◆ *Ending the Application* on page 298
- ◆ *Complete Program* on page 298

This chapter walks through an example in C++, `rates.cpp`. You will also find `Rates.java` in `yourCPLEXinstallation/examples/src/`. If your installation includes the .NET API of ILOG CPLEX, then you will also find the C#.NET implementation of this example in `Rates.cs` and the VB.NET implementation in `Rates.vb`.

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). The semi-continuous lower bound (sclb) must be finite. The upper bound (ub) need not be finite. The semi-continuous lower bound (sclb) must be greater than or equal to 0 (zero). An attempt to use a negative value for the semi-continuous lower bound (sclb) will raise an error.

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 *ILOG CPLEX Reference Manual of the C++ API*.

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 *ILOG CPLEX Reference Manual of the C++ API*.

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.

Semi-continuous variables can be specified in MPS and LP files. *Stating a MIP Problem* on page 246 tells you how to specify variables as semi-continuous.

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 Application

As in all C++ CPLEX applications, this program ends with a call to `IloEnv::end` to de-allocate the model and algorithm after they are no longer in use.

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

Complete Program

You can see the entire program online in the standard distribution of ILOG CPLEX at *yourCPLEXinstallation/examples/src/rates.cpp*. To run that example, you need a license for ILOG CPLEX.

You will also find *Rates.java* in *yourCPLEXinstallation/examples/src/*. If your installation includes the .NET API of ILOG CPLEX, then you will also find the C#.NET implementation of this example in *Rates.cs* and the VB.NET implementation in *Rates.vb*.

Using Piecewise Linear Functions in Optimization: a Transport Example

This chapter shows you how to represent piecewise linear functions in optimization with ILOG CPLEX and Concert Technology. In this chapter, you will find these topics:

- ◆ *Piecewise Linearity in ILOG CPLEX* on page 300
- ◆ *Describing the Problem* on page 304
- ◆ *Developing a Model* on page 307
- ◆ *Solving the Problem* on page 309
- ◆ *Displaying a Solution* on page 309
- ◆ *Ending the Application* on page 309

This chapter walks through an example in C++, `transport.cpp`. You will also find `Transport.java` in `yourCPLEXinstallation/examples/src/`. If your installation includes the .NET API of ILOG CPLEX, then you will also find the C#.NET implementation of this example in `Transport.cs` and the VB.NET implementation in `Transport.vb`.

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 also 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, $(-\infty, 4)$ and $[4, 5)$ and $[5, 7)$ and $[7, \infty)$, 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 constant, though it is different between segments. The points where the slope of the function changes are known as *breakpoints*. The piecewise linear function in Figure 16.1 has three breakpoints.

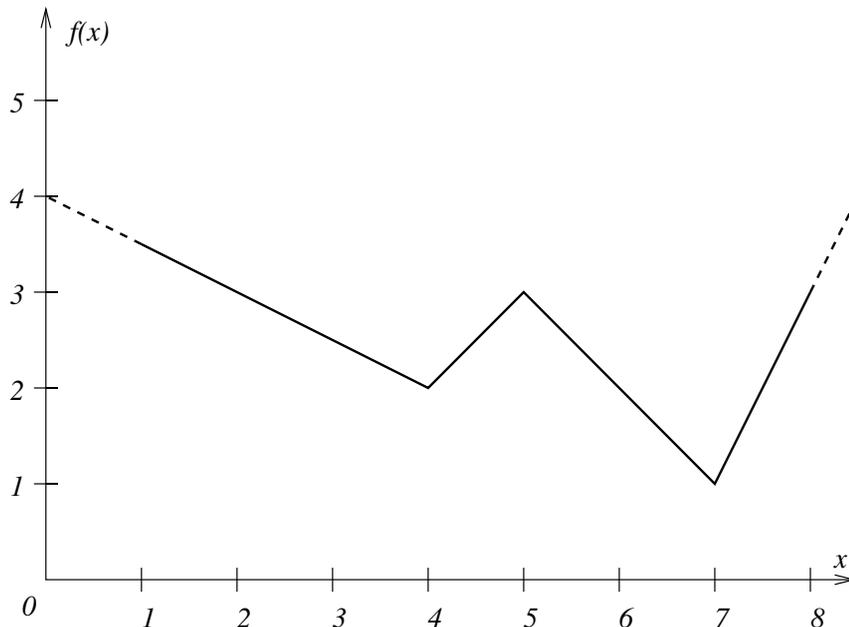


Figure 16.1 A piecewise linear function with breakpoints

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 independent 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 the endpoint of one segment and the initial point of the next segment 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 x -coordinate of the breakpoint where the step occurs is repeated in the array of the breakpoint.
- ◆ 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 in both segments associated with the step. For example, in Figure 16.2, at the breakpoint $x=3$, the points $(3, 1)$ and $(3, 3)$ are both admissible. Similarly, when $x = 5$, the points $(5, 4)$ and $(5, 5)$ are both admissible.

However, isolated points, as explained in *Isolated Points in Piecewise Linear Functions* on page 303, are not allowed, neither in continuous nor in discontinuous piecewise linear functions. In fact, only one step is allowed at a given point.

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);
```

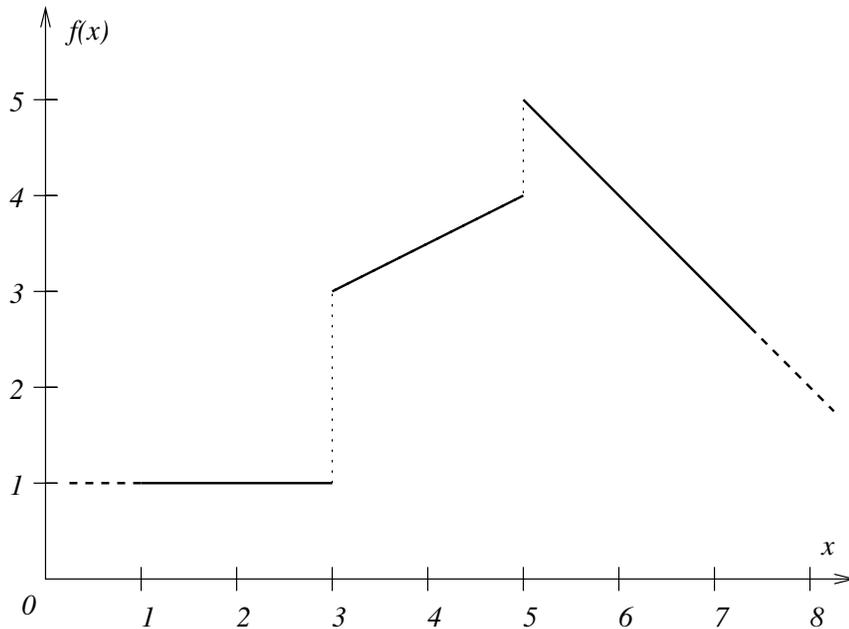


Figure 16.2 A discontinuous piecewise linear function with steps

Isolated Points in Piecewise Linear Functions

When you specify the same point more than twice as you declare a piecewise linear function, you inadvertently create an isolated point. ILOG CPLEX does not support isolated points. When it encounters an isolated point in the declaration of a piecewise linear function, ILOG CPLEX issues a warning and ignores the isolated point. An isolated point may appear as a visible point in the graph of a discontinuous piecewise linear function. For example, the point $(3, 2)$ would be an isolated point in Figure 16.2 and consequently ignored by ILOG CPLEX. Isolated points may also be less conspicuously visible; for example, if the height of a step in a discontinuous piecewise linear function is 0 (zero), the isolated point overlaps with an endpoint of two other segments, and consequently, the isolated point will be ignored by ILOG CPLEX.

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`).

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:

$$\begin{aligned} \text{supply}_0 &= 1000 \\ \text{supply}_1 &= 850 \\ \text{supply}_2 &= 1250 \end{aligned}$$

Each showroom has a fixed demand:

$$\begin{aligned} \text{demand}_0 &= 900 \\ \text{demand}_1 &= 1200 \\ \text{demand}_2 &= 600 \\ \text{demand}_3 &= 400 \end{aligned}$$

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 $\text{nbDemand} + \text{nbSupply}$ constraints that force all demands to be satisfied and all supplies to be shipped. Thus far, a model for our problem looks like this:

$$\begin{aligned} &\text{Minimize} && \sum_{i=0}^{\text{nbDemand}-1} \sum_{j=0}^{\text{nbSupply}-1} \text{cost}_{ij} \cdot x_{ij} \\ &\text{subject to} && \\ &&& \sum_{j=0}^{\text{nbSupply}-1} x_{ij} = \text{supply}_i \quad i = 0, \dots, \text{nbDemand} - 1 \\ &&& \sum_{i=0}^{\text{nbDemand}-1} x_{ij} = \text{demand}_j \quad j = 0, \dots, \text{nbSupply} - 1 \end{aligned}$$

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.

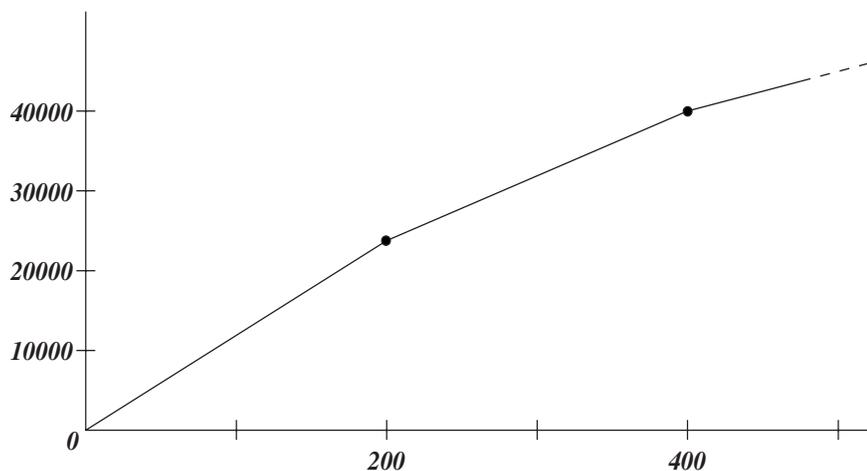


Figure 16.3 A concave piecewise linear cost function

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.

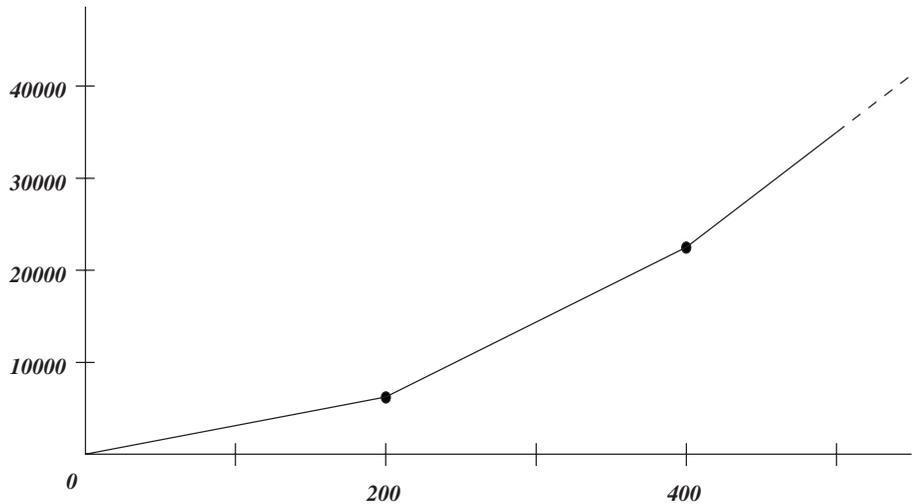


Figure 16.4 A convex piecewise linear cost function

Model with Varying Costs

With this additional consideration about costs varying according to quantity, our model now looks like this:

$$\begin{aligned} &\text{minimize} && \sum_{i=0}^{nbDemand-1} \sum_{j=0}^{nbSupply-1} y_{ij} \\ &\text{subject to} && \\ & && y_{ij} = f(x_{ij}) \text{ for } i = 0, \dots, nbDemand-1 \text{ and } j = 0, \dots, nbSupply-1 \end{aligned}$$

$$\sum_{j=0}^{nbSupply-1} x_{ij} = demand_i \quad \text{for } i = 0, \dots, nbDemand-1$$

$$\sum_{i=0}^{nbDemand-1} x_{ij} = supply_j \quad \text{for } j = 0, \dots, nbSupply-1$$

With this problem in mind, consider how to represent the data and model in Concert Technology.

Developing a Model

As in other examples in this manual, this application begins by creating an environment, an instance of `IloEnv`.

```
IloEnv env;
```

Within that environment, a model for this problem is created as an instance of `IloModel`.

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

Then constraints and an objective are added to the model. The following sections sketch these steps.

- ◆ *Representing the Data* on page 307
- ◆ *Adding Constraints* on page 307
- ◆ *Checking Convexity and Concavity* on page 308
- ◆ *Adding an Objective* on page 308

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);  
for(i = 0; i < nbSupply; i++){  
    x[i] = IloNumVarArray(env, nbDemand, 0, IloInfinity, ILOFLOAT);  
    y[i] = IloNumVarArray(env, nbDemand, 0, IloInfinity, ILOFLOAT);  
}
```

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++) {          // 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();
}

```

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, 200.0, 400.0),
                IloNumArray(env, 3, 30.0, 80.0, 130.0),
                0.0, 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, 200.0, 400.0),
                IloNumArray(env, 3, 120.0, 80.0, 50.0),
                0.0, 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));
obj.end();

```

Solving the Problem

The following lines create an algorithm (an instance of `IloCplex`) in an environment (an instance of `IloEnv`) and extract the model (an instance of `IloModel`) for that algorithm to find a solution.

```
IloCplex cplex(env);
cplex.extract(model);
cplex.exportModel("transport.lp");
cplex.solve();
```

Displaying a Solution

To display the solution, use the methods of `IloEnv` and `IloCplex`.

```
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 online in the standard distribution of ILOG CPLEX at `yourCPLEXinstallation/examples/src/transport.cpp`. To run this example, you need a license for ILOG CPLEX.

You will also find `Transport.java` in `yourCPLEXinstallation/examples/src/`. If your installation includes the .NET API of ILOG CPLEX, then you will also find the C#.NET implementation of this example in `Transport.cs` and the VB.NET implementation in `Transport.vb`.

Logical Constraints in Optimization

This chapter shows you how to represent logical constraints in ILOG CPLEX with Concert Technology. Concert Technology can automatically translate logical constraints into their transformed equivalent that the discrete (that is, MIP) or continuous (LP) optimizers of ILOG CPLEX can process efficiently in the C++, Java, or .NET APIs.

In the Callable Library, *indicator constraints* provide a similar facility. For more about that idea, see *Using Indicator Constraints* on page 317 in this manual.

In this chapter, you will learn:

- ◆ *What Are Logical Constraints?* on page 312
- ◆ *What Can Be Extracted from a Model with Logical Constraints?* on page 312
- ◆ *Which Nonlinear Expressions Can Be Extracted?* on page 314
- ◆ *Logical Constraints for Counting* on page 315
- ◆ *Logical Constraints as Binary Variables* on page 315
- ◆ *How Are Logical Constraints Extracted?* on page 315

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 Nonlinear Expressions Can Be Extracted?* on page 314.

What Can Be Extracted from a Model with Logical Constraints?

Much the same logical constraints are available in these APIs of ILOG CPLEX.

- ◆ *Logical Constraints in the C++ API* on page 312
- ◆ *Logical Constraints in the Java API* on page 313
- ◆ *Logical Constraints in the .NET API* on page 313

For similar facilities in the Callable Library, see *Using Indicator Constraints* on page 317.

Logical Constraints in the C++ API

In C++ applications, the class `IloCplex` can extract modeling objects to solve a wide variety of MIPs, as you see in *Solving the Model* on page 48, summarized in Table 1.1 on page 49. In fact, the C++ 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`
- ◆ `IloDiff`
- ◆ `==` that is, the equivalence relation

Among those extractable objects, `IloAnd`, `IloOr`, `IloNot`, and `IloDiff` can also be represented in your application by means of the overloaded C++ operators:

- ◆ `||` (for `IloOr`)
- ◆ `&&` (for `IloAnd`)
- ◆ `!` (for `IloNot`)

- ◆ `!=` that is, the exclusive-or relation (for `IloDiff`)

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.

Logical Constraints in the Java API

Of course, because the Java programming language does not support the overloading of operators as C++ does, overloaded logical operators are not supported in the Java API of Concert Technology. However, the Java class `IloCplexModeler` offers logical modeling facilities through methods, such as:

- ◆ `IloCplexModeler.and`
- ◆ `IloCplexModeler.or`
- ◆ `IloCplexModeler.not`
- ◆ `IloCplexModeler.ifThen`

Moreover, like their C++ counterparts, those extractable Java 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 Java application.

Logical Constraints in the .NET API

Similarly, the .NET API of Concert Technology supports logical constraints, though not operator overloading. The .NET class `Cplex` offers these overloaded logical methods:

- ◆ `Cplex.And`
- ◆ `Cplex.Or`

- ◆ `Cplex.Not`
- ◆ `Cplex.IfThen`

Again, those extractable .NET objects accept other linear constraints or logical constraints as their arguments, thus making it possible for you to combine linear constraints with logical constraints in expressions in your .NET applications.

Which Nonlinear Expressions Can Be Extracted?

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 any linear 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 numeric expressions or over a numeric expression and a constant in C++
- ◆ `IloMax` the maximum of an array of numeric expressions or over a numeric expression and a constant in C++
- ◆ `IloAbs` the absolute value of a numeric expression
- ◆ `IloPiecewiseLinear` the piecewise linear combination of a numeric 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` in a C++ application 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));
```

It is important to note here that only linear constraints can appear as arguments of logical constraints extracted by ILOG CPLEX. That is, quadratic constraints are **not** handled in logical constraints. Similarly, quadratic terms can **not** appear as arguments of logical expressions such as `IloMin`, `IloMax`, `IloAbs`, and `IloPiecewiseLinear`.

Logical Constraints for Counting

In many cases it is even unnecessary to allocate binary variables explicitly in order to gain the benefit of linear constraints within logical expressions. For example, 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.

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);
```

Logical Constraints as Binary Variables

Linear or logical constraints can appear as terms in numeric expressions. A linear constraint appearing as a term in a numeric expression 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));
```

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`.

How Are Logical Constraints Extracted?

Logical constraints are transformed automatically into equivalent linear formulations when they are extracted by an ILOG CPLEX algorithm. This transformation involves automatic creation by ILOG CPLEX of new variables and constraints. The transformation entails indicators as discussed in *Using Indicator Constraints* on page 317.

Using Indicator Constraints

This chapter introduces indicator constraints.

- ◆ *What Is an Indicator Constraint?* on page 317
- ◆ *Example: fixnet.c* on page 318
- ◆ *Indicator Constraints in the Interactive Optimizer* on page 318
- ◆ *What Are Indicator Variables?* on page 319
- ◆ *Restrictions on Indicator Constraints* on page 319
- ◆ *Best Practices with Indicator Constraints* on page 319

What Is an Indicator Constraint?

An *indicator constraint* is a way for a user of the Callable Library (C API) to express relationships among variables by identifying a binary variable to control whether or not a specified linear constraint is active. This feature is also available in the Interactive Optimizer, as explained in *Indicator Constraints in the Interactive Optimizer* on page 318.

Formulations using indicator constraints can be more numerically robust and accurate than conventional formulations involving so-called Big M data. Big M formulations use artificial data to turn on or turn off enforcement of a constraint. Big M formulations often exhibit trickle flow, and sometimes they behave in unstable ways.

In Concert Technology applications, ILOG CPLEX automatically uses indicator constraints for you when it encounters a constraint within an expression and when it encounters expressions which can be linearized, including the following:

- ◆ `IloAnd` or `Cplex.And`
- ◆ `IloOr` or `Cplex.Or`
- ◆ `IloNot` or `Cplex.Not`
- ◆ `IloIfThen` or `Cplex.IfThen`
- ◆ using a constraint as a binary variable

In Callable Library applications, you can invoke the routine `CPXaddindcontr` yourself to introduce indicator constraints in your model. To remove an indicator constraint that you have added, use the routine `CPXdelindconstr`.

Example: `fixnet.c`

For an example of indicator constraints in use, see `fixnet.c` among the examples distributed with the product. This example contrasts a model of a fixed-charge problem using indicator constraints with a Big M model of the same problem. That contrast shows how artificial data lead to an answer that is different from the result that the formulator of the model intended.

Indicator Constraints in the Interactive Optimizer

In the Interactive Optimizer, you can include indicator constraints among the usual linear constraints in LP-file format. You can also use the commands `enter` and `add` with indicator constraints. For example, you could declare `y` as a binary variable and enter the following:

```
constr01: y = 0 -> x1 + x2 + x3 = 0
```

This formulation of an indicator constraint is recommended instead of the following Big M formulation:

```
constr01: x1 + x2 + x3 - 1e+9 y <= 0 // not recommended
```

That Big M formulation relies on the `x` values summing to less than the Big M value (in this case, one billion). Such an assumption may cause numeric instability or undesirable solutions in certain circumstances, whereas a model with the indicator constraint, by contrast, introduces no new assumptions about upper bounds.

What Are Indicator Variables?

The binary variable introduced in an indicator constraint is known as an *indicator variable*. Usually, an indicator variable will also appear in the objective function or in other constraints. For example, in `fixnet.c`, the indicator variables `f` appear in the objective function to represent the cost of building an arc. In fact, an indicator variable introduced in one indicator constraint may appear again in another, subsequent indicator constraint.

Restrictions on Indicator Constraints

There are a few restrictions regarding indicator constraints:

- ◆ The constraint must be linear; a quadratic constraint is not allowed to have an indicator constraint.
- ◆ A lazy constraint cannot have an indicator constraint.
- ◆ A user-defined cut cannot have an indicator constraint.
- ◆ Only $z=0$ (zero) or $z=1$ (one) are allowed for the indicator variable because the indicator constraint implies that the indicator variable is binary.

ILOG CPLEX does not impose any arbitrary limit on the number of indicator constraints or indicator variables that you introduce, but there may be practical limits due to resources available on your platform.

Best Practices with Indicator Constraints

The following points summarize best practices with indicator constraints in Callable Library applications:

- ◆ Use indicator constraints when Big M values in the formulation cannot be reduced.
- ◆ Do not use indicator constraints if Big M can be avoided.
- ◆ Do not use indicator constraints if Big M is eliminated by preprocessing. Check the presolved model for Big M.
- ◆ If valid upper bounds on continuous variables are available, use them. Bounds strengthen LP relaxations. Bounds are used in a MIP for fixing and so forth.

Using Logical Constraints: Food Manufacture 2

Logical Constraints in Optimization on page 311, introduced features of ILOG CPLEX that transform parts of your problem automatically for you. This chapter shows you some of those features in use in a C++ application. 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 form 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. This chapter covers these topics:

- ◆ *Describing the Problem* on page 322
- ◆ *Representing the Data* on page 322
- ◆ *Developing the Model* on page 325
- ◆ *Using Logical Constraints* on page 327
- ◆ *Solving the Problem* on page 327
- ◆ *Ending the Program* on page 328

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 (v_1) or vegetable oil 2 (v_2) is blended in the product, then non vegetable oil 3 (o_3) 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 323
- ◆ *What Is Unknown?* on page 323
- ◆ *What Are the Constraints?* on page 324
- ◆ *What Is the Objective?* on page 325

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);
}
```

In those lines, the type `NumVarMatrix` is defined as:

```
typedef IloArray<IloNumVarArray> NumVarMatrix;
```

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 322, 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: Alternatively, you could use semi-continuous 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, where `profit` is defined as `IloExpr profit(env);`

```
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 324), like this:

```
IloExpr profit(env);
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 324. 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 English 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 311 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();
```

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 330
- ◆ *Understanding the Data File* on page 330
- ◆ *Reading the Data* on page 331
- ◆ *Creating Variables* on page 331
- ◆ *Stating Precedence Constraints* on page 332
- ◆ *Stating Resource Constraints* on page 332
- ◆ *Representing the Piecewise Linear Cost Function* on page 332
- ◆ *Transforming the Problem* on page 333
- ◆ *Solving the Problem* on page 334

This chapter walks through the C++ implementation. You can compare the Java implementation of the same model, using logical constraints, piecewise linear functions, and aggressive MIP emphasis in `Etsp.java`, and the C#.NET implementation in `Etsp.cs`, and the VB.NET implementation in `Etsp.vb` as well.

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`.

The data of this example consists of arrays and arrays of arrays (that is, matrices).

One array of arrays represents the *resources* required for each activity of a job. For example, `job0` entails eight activities, and those eight activities require the following ordered list of resources:

```
1, 3, 4, 1, 2, 4, 2, 4
```

A second array of arrays represents the *duration* required for each activity of a job. For `job0`, the following ordered list represents the duration of each activity:

```
41, 32, 72, 65, 53, 35, 53, 2
```

In other words, `job0` requires `resource1` for a duration of 41 time units; then `job0` requires `resource3` for 32 time units, and so forth.

There is also an array representing the due date of each job. That is, `array[i]` specifies the due date of `jobi`.

To represent the penalty for the early completion of each job, there is an array of penalties.

Likewise, to represent the penalty for late completion of each job, there is an array of penalties for tardiness.

Reading the Data

The first part of this application reads data from a file and fills matrices:

```
IloEnv env;

IntMatrix   activityOnAResource (env);
NumMatrix   duration (env);
IloNumArray jobDueDate (env);
IloNumArray jobEarlinessCost (env);
IloNumArray jobTardinessCost (env);

f >> activityOnAResource;
f >> duration;
f >> jobDueDate;
f >> jobEarlinessCost;
f >> jobTardinessCost;

IloInt nbJob      = jobDueDate.getSize();
IloInt nbResource = activityOnAResource.getSize();
```

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.

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. The type `NumVarMatrix` is defined as `typedef IloArray<IloNumVarArray> NumVarMatrix;`

```
NumVarMatrix s(env, nbJob);
for(j = 0; j < nbJob; j++){
    s[j] = IloNumVarArray(env, nbResource, 0.0, 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:

$$s_1 \geq s_2 + d_2 \text{ or } s_2 \geq s_1 + d_1$$

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 20.1. 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));
```

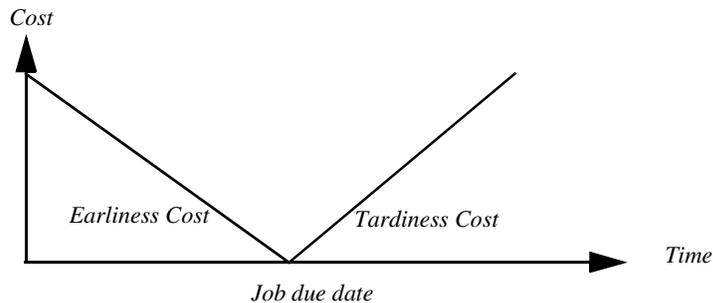


Figure 20.1 Earliness and Tardiness as Piecewise Linear Cost Function

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.

Solving the Problem

An emphasis on finding hidden feasible solutions has proven particularly effective for this problem so this example makes that selection by setting the `MIPEmphasis` parameter to 4.

You can see the entire example online in the standard distribution of ILOG CPLEX at *yourCPLEXinstallation/examples/src/etsp.cpp*. Implementations of the same model, using the same features of ILOG CPLEX, are available as `Etsp.java`, `Etsp.cs`, and `Etsp.vb` as well.

Using Column Generation: a Cutting Stock Example

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).

This chapter walks through an example in C++, `cutstock.cpp`. You will also find `CutStock.java` in *yourCPLEXinstallation/examples/src/*. If your installation includes the .NET API of ILOG CPLEX, then you will also find the C#.NET implementation of this example in `CutStock.cs` and the VB.NET implementation in `CutStock.vb`.

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 generation 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 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 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 other 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 *Reference Manual of the C++ API*, 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 of fixed length (the stock). Strips are cut from these rolls. 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 of strips. 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.

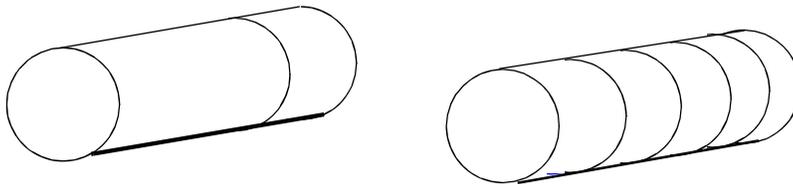


Figure 21.1 Two different patterns from a roll of stock

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 :

$$\begin{aligned} &\text{Minimize} && \sum_j X_j \\ &\text{subject to} && \sum_j A_{ij} X_j \geq d_i \text{ with } X_j \geq 0 \end{aligned}$$

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 since it probably uses too many rolls.

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 with the help of a subproblem. This subproblem is defined to identify the coefficients of a new column of the master problem with minimal reduced cost. With π_i as the vector of the dual variables of the current solution of the master problem, the subproblem is defined like this:

$$\begin{aligned} & \text{Minimize } 1 - \sum_i \pi_i A_i \\ & \text{subject to } \sum_i W_i A_i \leq W \end{aligned}$$

where W is the width of a roll of stock and the entries A_i are the modeling variables of the subproblem. Their solution values will be the coefficients of the new column to be added to the master model if a solution with a negative objective function is found for the subproblem. Consequently, the variables A_i must be nonnegative integers.

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 strips 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 to represent the master model. Later, through its modifications, another model `patGen` is built to generate the new columns. That is, `patGen` represents the subproblem.

The first model `cutOpt`, an instance of `IloModel`, is declared like this:

```
IloModel cutOpt (env);
```

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 *Reference Manual of the C++ API*.)

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`). 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 for the submodel in this line:

```
patGen.add(IloScalProd(size, Use) <= rollWidth);
```

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 master model is initialized with one variable for each size. Each such variable represents the pattern of cutting a roll into as many strips of that size as possible. These variables are stored as they are created in the array `Cut` by the following loop:

```
IloInt nWdth = size.getSize();
for (j = 0; j < nWdth; j++)
    Cut.add(IloNumVar(RollsUsed(1) + Fill(1)(int(rollWidth / size[j]))));
```

Consequently, the variable `Cut[j]` will have an objective coefficient of 1 (one) and only one other nonzero coefficient (`rollWidth/size[j]`) for constraint `Fill[j]`. Later, in the column generation loop, new variables will be added. Those variables will have coefficients defined by the solution vectors of the subproblem stored in the array `newPatt`.

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 with coefficient

`newPatt[i]` for constraint `Fill[i]`; they also 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.

Changing the Type of a Variable

After the column-generation phase terminates, an integer solution to the master problem must be found. To do so, the type of the variables must be changed from continuous to integer.

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`) 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` for 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 submodel of the cutting stock problem is represented by the model `patGen` in this example. This pattern generator `patGen` (in contrast to `cutOpt`) is defined by the 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, nWidth, 0, IloInfinity, ILOINT);
patGen.add(IloScalProd(size, Use) <= rollWidth);
```

Changing the Objective Function

After the dual solution vector of the master model is available, the objective function of the subproblem is adjusted by a call to the method `IloObjective::setLinearCoefs`, like this:

```
ReducedCost.setLinearCoefs(Use, price);
```

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 continuous relaxation of the column-generation problem. In other words, the application drops the requirement for integrality of the variables while the columns are generated. After all columns have been generated for the continuous relaxation, the application keeps the variables generated so far, changes their type to integer, and solves the resulting integer problem.

As you've seen, this example manages two models of the problem, `cutOpt` and `patGen`. Likewise, it uses two algorithms (that is, two instances of `IloCplex`) to solve them.

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 the model `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.setLinearCoefs(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 of the subproblem is nonnegative within the 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 online in the standard distribution of ILOG CPLEX at *yourCPLEXinstallation/examples/src/cutstock.cpp*.

You will also find *CutStock.java* in *yourCPLEXinstallation/examples/src/*. If your installation includes the .NET API of ILOG CPLEX, then you will also find the C#.NET implementation of this example in *CutStock.cs* and the VB.NET implementation in *CutStock.vb*.

Part V

Infeasibility and Unboundedness

The topics discussed in *Continuous Optimization* on page 159 and *Discrete Optimization* on page 243 often contained the implicit assumption that a bounded feasible solution to your model actually exists. This part of the manual discusses what steps to try when the outcome of an optimization is a declaration that your model is either:

- *infeasible*; that is, no solution exists that satisfies all the constraints, bounds, and integrality restrictions;
- or
- *unbounded*; that is, the objective function can be made arbitrarily large; a more careful definition of unbounded is provided in *What Is Unboundedness?* on page 350.

Infeasibility and unboundedness are closely related topics in optimization theory, and therefore certain of the concepts for one will have direct relation to the other. This part contains:

- ◆ *Preprocessing and Feasibility* on page 347
- ◆ *Managing Unboundedness* on page 349
- ◆ *Diagnosing Infeasibility by Refining Conflicts* on page 353
- ◆ *Repairing Infeasibilities with FeasOpt* on page 371

As you know, ILOG CPLEX can provide solution information about the models that it optimizes. For infeasible outcomes, it reports values that you can analyze to determine what

in your problem formulation caused this result. In certain situations, you can then alter your problem formulation or change ILOG CPLEX parameters to achieve a satisfactory solution.

Infeasibility can arise from various causes, and it is not possible to automate procedures to deal with those causes entirely without input or intervention from the user. For example, in a shipment model, infeasibility could be caused by insufficient supply, or by an error in demand, and it is likely that the optimizer will tell the user only that the mismatch exists. The formulator of the model has to make the ultimate judgment of what the actual error is. However, there are ways to try to narrow down the investigation or even provide some degree of automatic repair.

ILOG CPLEX provides tools to help you analyze the source of the infeasibility in a model. Those tools include the **conflict refiner** for detecting minimal sets of mutually contradictory bounds and constraints, and **FeasOpt** for repairing infeasibilities.

Preprocessing and Feasibility

ILOG CPLEX preprocessing may declare a model infeasible before the selected optimization algorithm begins. This early declaration saves considerable execution time in most cases. When this declaration is the outcome of preprocessing, it is important 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 righthand side (RHS) of the constraints 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.

Managing Unboundedness

This chapter discusses the tactics you can use to diagnose the cause of an unbounded outcome in the optimization of your model. It also suggests ways to avoid an unbounded outcome.

- ◆ *What Is Unboundedness?* on page 350
- ◆ *Avoiding Unboundedness* on page 350
- ◆ *Diagnosing Unboundedness* on page 351

What Is Unboundedness?

Any class of model, continuous or discrete, linear or quadratic, has the potential to result in a solution status of *unbounded*. An unbounded discrete model must have a continuous relaxation that is also unbounded. Therefore, the discussion here will assume that you will first relax any discrete elements, and thus you are dealing with an unbounded continuous optimization problem, when trying to diagnose the cause.

Note: The reverse of that observation that an unbounded discrete model necessarily having an unbounded continuous relaxation is not necessarily the case: a discrete optimization model may have an unbounded continuous relaxation and yet have a bounded optimum.

A declaration of unboundedness means that ILOG CPLEX has determined that the model has an *unbounded ray*. That is, given any feasible solution x with objective z , a multiple of the unbounded ray can be added to x to give a feasible solution with objective $z-1$ (or $z+1$ for maximization models). Thus, if a feasible solution exists, then the optimal objective is unbounded.

When a model is declared unbounded, ILOG CPLEX has **not** necessarily concluded that a feasible solution exists. Users can call methods or routines to determine whether ILOG CPLEX has also concluded that the model has a feasible solution.

- ◆ In Concert Technology, call one of these methods:
 - `isDualFeasible`
 - `isPrimalFeasible`
 - try/catch the exception
- ◆ In the Callable Library, call the routine `CPXsolninfo`.

Avoiding Unboundedness

Unboundedness can be viewed as an under-constrained condition; such an outcome can be from a modeler forgetting to include one or more constraints in the model. Therefore carefully checking that your problem **formulation is complete** is a good first step in diagnosing unboundedness.

The default variable type in CPLEX has a lower bound of 0 (zero) and an upper bound of infinity. If you declare a variable to be of type `Free`, its lower bound is negative infinity instead of 0 (zero). A model can not be unbounded unless one or more of the variables has either of these infinite bounds. Therefore, one straightforward tactic in avoiding unboundedness is to **assign finite bounds** to every variable in your model; if no variable can go on an unbounded ray to infinity, then your model can not be unbounded.

Other forms of avoiding under-constrained conditions, such as **adding a constraint** that limits the sum of all variables, are also possible.

If an unbounded solution is not possible in the physical system you are modeling, then adding finite lower and upper bounds or adding other constraints may represent something realistic about the system that is worth expressing in the model anyway. However, great care should be taken to **assign meaningful bounds**, in cases where it is not possible to be certain what the actual bounds should be. If you happen to select bounds that are tighter than an optimal solution would obtain, then you can get a solution of worse objective function value than you want. On the other hand, picking extremely large numbers for bounds (just to be safe) carries some risk, too: on a finite-precision computer, even a bound of one billion may introduce numeric instability and cause the optimizer to solve less rapidly or not to converge to a solution at all, or may result in solutions that satisfy tolerances but contain small infeasibilities.

Diagnosing Unboundedness

You may be able to diagnose the cause of unboundedness by **examining the output** from the optimizer that made the determination. For example, if the presolve step at the beginning of optimization made a series of reductions and then stopped with a message like this:

```
Primal unbounded due to dual bounds, variable 'x1'.
```

it makes sense to look at your formulation, paying particular attention to variable x_1 and its interactions. Perhaps x_1 never intersects less-than-or-equal-to constraints with a positive coefficient (or, greater-than-or-equal-to constraints with a negative coefficient), and by inspection you can see that nothing prevents x_1 from going to infinity.

Similarly, the primal simplex optimizer may terminate with a message like this:

```
Diverging variable = x2
```

In such a case, you should focus attention on x_2 . (The dual simplex and barrier optimizers work differently than primal; they do not determine unboundedness in this way.) Unfortunately, the variable which is reported in one of these ways may or may not be a direct cause of the unboundedness, because of the many algebraic manipulations performed by the optimizer along the way.

An approach to diagnosis that is related to the technique discussed in *Avoiding Unboundedness* on page 350 is to **temporarily assign finite bounds** to all variables. By solving the modified model and determining which variables have solution values at these artificial bounds, you may be able to trace the cause through the constraints involving those variables.

Since an unbounded outcome means that an unbounded ray has been determined to exist, one approach to diagnosis is to **display this ray**. In Concert Technology, use the method

`getRay`; in the Callable Library use the advanced routine `CPXgetray`. The relationship of the variables in this ray may give you guidance as to the cause of unboundedness.

If you are familiar with LP theory, then you might consider **transforming your model** to the associated dual formulation. This transformation can be accomplished, for example, by writing out the model in DUA format and then reading it back in. (See the *ILOG CPLEX Reference Manual for File Formats* for a description of DUA as a file format.) The dual model of an unbounded model will be infeasible. And that means that you can use the conflict refiner to reduce the infeasible model to a minimal conflict. (See *Diagnosing Infeasibility by Refining Conflicts* on page 353 for more about the conflict refiner.) It is possible that the smaller model will allow you to identify the source of the (dual) infeasibility more easily than the full model allows.

Diagnosing Infeasibility by Refining Conflicts

This chapter tells you about the *conflict refiner*, a feature of ILOG CPLEX for diagnosing the cause of infeasibility in a model, whether continuous or discrete, whether linear or quadratic.

- ◆ *What Is a Conflict?* on page 353
- ◆ *What a Conflict Is Not* on page 354
- ◆ *How to Invoke the Conflict Refiner* on page 355
- ◆ *How a Conflict Differs from an IIS* on page 355
- ◆ *Meet the Conflict Refiner in the Interactive Optimizer* on page 356
- ◆ *Using the Conflict Refiner in an Application* on page 365

What Is a Conflict?

A *conflict* is a set of mutually contradictory constraints and bounds within a model. Given an infeasible model, ILOG CPLEX can identify conflicting constraints and bounds within it. ILOG CPLEX refines an infeasible model by examining elements that can be removed from

the conflict to arrive at a minimal conflict. A conflict smaller than the full model may make it easier for the user to analyze the source of infeasibilities in the original model.

If the model happens to contain multiple independent causes of infeasibility, it may be necessary for the user to repair one cause and then repeat the process with a further refinement.

What a Conflict Is Not

Information about the necessary magnitude of change to data values, in order to gain feasibility, is not available from a conflict. The algorithms for detecting and refining conflicts do their work by including or removing a constraint or bound in trial solutions, not by varying the data of those entities. For that kind of insight, or for an approach to automatic repair of infeasibility, the FeasOpt feature, discussed in *Repairing Infeasibilities with FeasOpt* on page 371, is more appropriate.

ILOG CPLEX refines conflicts only among the constraints and bounds in your model. It disregards the objective function while it is refining a conflict. In particular, if you have set a MIP cutoff value with the idea that the cutoff value will render your model infeasible, and then you apply the conflict refiner, you will not achieve the effect you expect. In such a case, you should add one or more explicit constraints to enforce the restriction you have in mind. In other words, add constraints rather than attempt to enforce a restriction through these parameters:

- `CutLo` or `CutUp` in Concert Technology (**not** recommended to enforce infeasibility)
- `CPX_PARAM_CUTLO` or `CPX_PARAM_CUTUP` in the Callable Library (**not** recommended to enforce infeasibility)
- `mip tolerance lowercutoff` or `uppercutoff` in the Interactive Optimizer (**not** recommended to enforce infeasibility)

How to Invoke the Conflict Refiner

Table 24.1 summarizes the methods and routines that invoke the conflict refiner, depending on the component or API that you choose.

Table 24.1 Conflict Refiner

API or Component	Invoke Conflict Refiner	Access Results	Save Results
Concert Technology for C++ Users	<code>IloCplex::refineConflict</code>	<code>getConflict</code>	<code>writeConflict</code>
Concert Technology for Java Users	<code>IloCplex.refineConflict</code>	<code>getConflict</code>	<code>writeConflict</code>
Concert Technology for .NET Users	<code>Cplex.RefineConflict</code>	<code>GetConflict</code>	<code>WriteConflict</code>
Callable Library	<code>CPXrefineconflict</code> <code>CPXrefineconflicttext</code>	<code>CPXgetconflict</code> <code>CPXgetconflicttext</code>	<code>CPXclpwrite</code>
Interactive Optimizer	<code>conflict</code>	<code>display conflict all</code>	<code>write file.clp</code>

The following sections explain more about these methods and routines.

How a Conflict Differs from an IIS

In some ways a conflict resembles an irreducibly inconsistent set (IIS). Detection of an IIS among the constraints of a model is a standard methodology in the published literature; an IIS finder has long been available as a tool within ILOG CPLEX. Both tools (conflict refiner and IIS finder) attempt to identify an infeasible subproblem of a provably infeasible model.

However, a conflict is more general than an IIS. The IIS finder is applicable only to continuous LP models, whereas the conflict refiner is capable of doing its work on any type of problem, including mixed integer models or models containing quadratic elements.

Also, you can specify one or more *groups* of constraints for a conflict; a group will either be present together in the conflict, or else will not be part of it at all.

You can also assign numeric *preference* to a constraint or to groups of constraints. In the case of an infeasible model that has more than one possible conflict, the preferences you assign will guide the tool toward detecting the conflict you want. Preferences allow you to specify aspects of the model that may otherwise be difficult to encode.

While the conflict refiner usually will deliver a smaller set of constraints to consider than the IIS finder will, the methods are different enough that the reverse can sometimes be true. The fact that the IIS finder implements a standard methodology may weigh toward its use in some situations. Otherwise, the conflict refiner can be thought of as usually doing

everything the IIS finder can, and often more. In fact, you might think of the conflict refiner as an extension and generalization of the IIS finder.

Meet the Conflict Refiner in the Interactive Optimizer

You can get acquainted with the conflict refiner in the Interactive Optimizer. Certain features of the conflict refiner, namely, preferences and groups, are available only through an application of the Callable Library or Concert Technology. Those additional features are introduced in *Using the Conflict Refiner in an Application* on page 365. This introduction of the conflict refiner through the Interactive Optimizer covers these topics:

- ◆ *A Model for the Conflict Refiner* on page 356
- ◆ *Optimizing the Example* on page 357
- ◆ *Interpreting the Results and Detecting Conflict* on page 357
- ◆ *Displaying a Conflict* on page 358
- ◆ *Interpreting Conflict* on page 359

A Model for the Conflict Refiner

Here's a simplified resource allocation problem to use as a model in the Interactive Optimizer. Either you can create a file containing these lines and read the file into the Interactive Optimizer by means of this command:

```
read filename
```

or you can use the `enter` command, followed by a name for the problem, followed by these lines:

```
Minimize
  obj: cost
Subject To
  c1:  - cost + 80 x1 + 60 x2 + 55 x3 + 30 x4 + 25 x5 + 80 x6 + 60 x7 + 35 x8
      + 80 x9 + 55 x10 = 0
  c2:  x1 + x2 + 0.8 x3 + 0.6 x4 + 0.4 x5 >= 2.1
  c3:  x6 + 0.9 x7 + 0.5 x8 >= 1.2
  c4:  x9 + 0.9 x10 >= 0.8
  c5:  0.2 x2 + x3 + 0.5 x4 + 0.5 x5 + 0.2 x7 + 0.5 x8 + x10 - service = 0
  c6:  x1 + x6 + x9 >= 1
  c7:  x1 + x2 + x3 + x6 + x7 + x9 >= 2
  c8:  x2 + x3 + x4 + x5 <= 0
  c9:  x4 + x5 + x8 <= 1
  c10: x1 + x10 <= 1
Bounds
  service >= 3.2
Binaries
  x1 x2 x3 x4 x5 x6 x7 x8 x9 x10
End
```

This simple model, for example, might represent a project-staffing problem. In that case, the ten binary variables could represent employees who could be assigned to duty.

The first constraint defines the cost function. In this example, the objective is to minimize the cost of salaries. The next three constraints (c2, c3, c4) represent three nonoverlapping skills that the employees must cover to varying degrees of ability. The fifth constraint represents some additional quality metric (perhaps hard to measure) that most or all of the employees can contribute to. It is called customer service in this example. That variable has a lower bound to make sure of a certain predefined minimum level of 3.2.

The remaining constraints represent various work rules that reflect either policy that must be followed or practical guidance based on experience with this work force. Constraint c6, for example, dictates that at least one person with managerial authority be present. Constraint c7 requires at least two senior personnel be present. Constraint c8 indicates that several people are scheduled for off-site training during this period. Constraint c9 recognizes that three individuals are not productive together. Constraint c10 prevents two employees who are married to each other from working in this group in the same period, since one is a manager.

Optimizing the Example

If you apply the `optimize` command to this example, you will see these results:

```
Row 'c8' infeasible, all entries at implied bounds.  
Presolve time = 0.00 sec.  
MIP -- Integer infeasible.  
Current MIP best bound is infinite.  
Solution time = 0.00 sec. Iterations = 0 Nodes = 0
```

Interpreting the Results and Detecting Conflict

The declaration of infeasibility comes from presolve. In fact, presolve has already performed various reductions by the time it detects the unresolvable infeasibility in constraint c8. This information by itself is unlikely to provide any useful insights about the source of the infeasibility, so try the conflict refiner, by entering this command:

```
conflict
```

Then you will see results like these:

```
Refine conflict on 14 members...
Iteration  Max Members  Min Members
          1             11             0
          2             9             0
          3             7             0
          4             2             0
          5             2             1
          6             2             2
Minimal conflict:  2 linear constraint(s)
                  0 lower bound(s)
                  0 upper bound(s)
Conflict computation time = 0.00 sec.  Iterations = 6
```

The first line of output mentions 14 members; this total represents constraints, lower bounds, and upper bounds that may be part of the conflict. There are ten constraints in this model; there are two continuous variables with lower and upper bounds that represent the other four members to be considered. Because binary variables are not reasonable candidates for bound analysis, the Interactive Optimizer treats the bounds of only the variables `cost` and `service` as potential members of the conflict. If you want all bounds to be candidates, you could instead declare the binary variables to be general integer variables with bounds of [0,1]. (Making that change in this model would likely result in the conflict refiner suggesting that one of the binary variables should take a negative value.) On some models, allowing so much latitude in the bounds may cause the conflict refiner to take far longer to arrive at a minimal conflict.

Displaying a Conflict

As you can see in the log displayed on the screen, the conflict refiner works to narrow the choices until it arrives at a conflict containing only two members. Since the conflict is small in this simplified example, you can see it in its entirety by entering this command:

```
display conflict all
```

```
Minimize
  obj:
Subject To
  c2: x1 + x2 + 0.8 x3 + 0.6 x4 + 0.4 x5 >= 2.1
  c8: x2 + x3 + x4 + x5 <= 0
Bounds
  0 <= x1 <= 1
  0 <= x2 <= 1
  0 <= x3 <= 1
  0 <= x4 <= 1
  0 <= x5 <= 1
Binaries
  x1 x2 x3 x4 x5
```

In a larger conflict, you can selectively display constraints or bounds on variables by using these commands to specify a range of rows or columns:

```
display conflict constraints
display conflict variables
```

You can also write the entire conflict to a file in LP-format to browse later by using the command (where *modelname* is the name you gave the problem):

```
write modelname.clp
```

Interpreting Conflict

In those results, you can see that c8, the constraint mentioned by presolve, is indeed a fundamental part of the infeasibility, as it directly conflicts with one of the skill constraints. In this example, with so many people away at training, the skill set in c2 cannot be covered. Perhaps it would be up to the judgment of the modeler or management to decide whether to relax the skill constraint or to reduce the number of people who will be away at training during this period, but something must be done for this model to have a feasible solution.

Deleting a Constraint

For the sake of explanation, assume that a decision is made to cancel the training in this period. To implement that decision, try entering this command:

```
change delete constraint c8
```

Now re-optimize. Unfortunately, even removing c8 does not make it possible to reach an optimum, as you can see from these results of optimization:

```
Constraints 'c5' and 'c9' are inconsistent.
Presolve time =    0.00 sec.
MIP -- Integer infeasible.
Current MIP best bound is infinite.
Solution time =    0.00 sec.  Iterations = 0  Nodes = 0
```

Perhaps presolve has identified a source of infeasibility, but if you run the `conflict` command again, you will see these results:

```
Refine conflict on 13 members...
  Iteration  Max Members  Min Members
          1             12             0
          2              9             0
          3              6             0
          4              4             0
          5              3             0
          6              3             1
          7              3             2
          8              3             3
Minimal conflict:  2 linear constraint(s)
                  1 lower bound(s)
                  0 upper bound(s)
Conflict computation time =    0.00 sec.  Iterations = 8
```

Now view the entire conflict with this command:

```
display conflict all
```

```
Minimize
  obj:
Subject To
  c5:      0.2 x2 + x3 + 0.5 x4 + 0.5 x5 + 0.2 x7 + 0.5 x8 + x10 - service = 0
  c       x4 + x5 + x8 <= 1
  sum_eq: 0.2 x2 + x3 + 0.5 x4 + 0.5 x5 + 0.2 x7 + 0.5 x8 + x10 - service = 0
Bounds
  0 <= x2 <= 1
  0 <= x3 <= 1
  0 <= x4 <= 1
  0 <= x5 <= 1
  0 <= x7 <= 1
  0 <= x8 <= 1
  0 <= x10 <= 1
          service >= 3.2
Binaries
  x2 x3 x4 x5 x7 x8 x10
```

Understanding a Conflict Report

The constraints mentioned by `presolve` are part of the minimal conflict detected by the conflict refiner. The additional information provided by this conflict is that the lower bound on `service` quality could also be considered for modification to achieve feasibility: with only one among employees 4, 5, and 8 permitted, any of whom contribute 0.5 to the quality metric, the lower bound on `service` can not be achieved. Unlike a binary variable, where it would make little sense to adjust either of its bounds to achieve feasibility, the bounds on a continuous variable like `service` may be worth scrutiny.

The other information this Conflict provides is that no change of the upper bound on `service`, currently infinity, could aid toward feasibility; perhaps that is already obvious, but even a finite upper bound would not be part of this conflict (as long as it is larger than the lower bound of 3.2).

Summing Equality Constraints

Note the additional constraint provided in this conflict: `sum_eq`. It is a sum of all the equality constraints in the conflict. In this case, there is only one such constraint; sometimes when there are more, an imbalance will become quickly apparent when positive and negative terms cancel.

Changing a Bound

Again, for the sake of the example, assume that it is decided after consultation with management to repair the infeasibility by reducing the minimum on the `service` metric, on the grounds that it is a somewhat arbitrary metric anyway. A minimal conflict does not directly tell you the magnitude of change needed, but in this case it can be quickly determined by examination of the minimal conflict that a new lower bound of 2.9 could be achievable; select 2.8, to be safe. Modify the model by entering this command:

change bound service lower 2.8

and re-optimize. Now at last the model delivers an optimum:

```
Tried aggregator 1 time.
MIP Presolve eliminated 9 rows and 12 columns.
MIP Presolve modified 16 coefficients.
All rows and columns eliminated.
Presolve time = 0.00 sec.
Integer optimal solution: Objective = 3.3500000000e+02
Solution time = 0.00 sec. Iterations = 0 Nodes = 0
```

Displaying the solution indicates that employees {2,3,5,6,7,10} are used in the optimal solution.

Adding a Constraint

A natural question is why so many employees are needed. Look for an answer by adding a constraint limiting employees to five or fewer, like this:

```
add
x1+x2+x3+x4+x5+x6+x7+x8+x9+x10 <= 5
end
optimize
```

As you might expect, the output from the optimizer indicates the current solution is incompatible with this new constraint, and indeed no solution to this what-if scenario exists at all:

```
Warning: MIP start values are infeasible.
Retaining MIP start values for possible repair.
Row 'c11' infeasible, all entries at implied bounds.
Presolve time = 0.00 sec.
MIP -- Integer infeasible.
Current MIP best bound is infinite.
Solution time = 0.00 sec. Iterations = 0 Nodes = 0
```

Constraint c11, flagged by presolve, is the newly added constraint, not revealing very much. To learn more about why c11 causes trouble, run `conflict` again, and view the minimal conflict with the following command again:

```
display conflict all
```

You will see the following conflict:

```
Minimize
obj:
Subject To
c2: x1 + x2 + 0.8 x3 + 0.6 x4 + 0.4 x5 >= 2.1
c3: x6 + 0.9 x7 + 0.5 x8 >= 1.2
c4: x9 + 0.9 x10 >= 0.8
c11: x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 <= 5
      (omitting the listing of binary variables' bounds)
```

The constraints in conflict with this new limitation are all of the skill requirements. When viewed in this light, the inconsistency is easy to spot: one employee is obviously needed for constraint c4, two are needed for c3, and a simple calculation reveals that three are needed for c2. Since there is no overlap in the skill sets, five employees are too few.

Unless management or the formulator of the model is willing to compromise about the skills, (for example, to relax the righthand side of any of these constraints), constraint c11 needs to be taken out again, since it is unrealistic to get by with only five employees:

```
change delete constraint c11
```

This change results in a model with an optimal cost of 335, using six employees.

Changing Bounds on Cost

No better cost is possible in this formulation. Still, you may wonder, "Why not?" To try yet another scenario, instead of limiting the number of employees, try focusing on the cost by changing the upper bound of the cost to 330, like this:

```
change bound cost upper 330
optimize
conflict
display conflict all
```

This series of commands again renders the model infeasible and shows a minimal conflict:

```
Subject To
c1:      - cost + 80 x1 + 60 x2 + 55 x3 + 30 x4 + 25 x5 + 80 x6 + 60 x7
          + 35 x8 + 80 x9 + 55 x10 = 0
c2:      x1 + x2 + 0.8 x3 + 0.6 x4 + 0.4 x5 >= 2.1
c3:      x6 + 0.9 x7 + 0.5 x8 >= 1.2
c5:      0.2 x2 + x3 + 0.5 x4 + 0.5 x5 + 0.2 x7 + 0.5 x8 + x10 - service
          = 0
c9:      x4 + x5 + x8 <= 1
Bounds
-Inf <= cost <= 330
service >= 2.9
```

The upper bound on `cost` is, of course, expected to be in the conflict, so relaxing it would merely put the scenario back the way it was. The constraint c1 defines `cost`, so unless there is some unexpected latitude in setting salaries, no relief will be found there. Constraints c2 and c3 represent two skill requirements, previously judged beyond negotiation, and constraint c5 represents service quality, already compromised a bit. That rough analysis leaves c9, the requirement not to use three particular employees together.

Relaxing a Constraint

How much is it costing to maintain this rule? Consider asking them to work productively pairwise, if not all three, and relax the upper limit of this constraint, like this:

```
change rhs c9 2
optimize
```

The model is now restored to feasibility, and the new optimum has an overall cost of 310, a tangible improvement of 25 over the previous optimum, using employees {2,3,5,6,8,10}; employee 7 has been removed in favor of employee 8. Is that enough monetary benefit to offset whatever reasons there were for separating employees 4 and 8? That is not a decision that can be made here; but at least this model provides some quantitative basis toward making that decision. Additionally, a check of the `service` variable shows that its solution value is back up to 3.2, a further benefit from relaxing constraint `c9`. Perhaps this decision should have been made sooner, the first time constraint `c9` appeared in a conflict.

The solution of 310 could be investigated further by changing the upper bound of `cost` to be 305, for example. The conflict resulting from this change consists of the skills constraint plus the constraint requiring at least one manager on duty. At this point, the analysis has reached a conclusion, unless management or the model formulator wishes to challenge the policy.

More about the Conflict Refiner

Presolve proved the infeasibility of that simplified example in *A Model for the Conflict Refiner* on page 356. However, a minimal conflict can be refined from an infeasible model regardless of how the infeasibility was found. The infeasibility may have been proven by presolve, by the continuous optimizers, or by the mixed integer optimizer.

A minimal conflict on a nontrivial model can take longer to refine than the associated optimization algorithm would have taken either to prove the infeasibility or to solve a similar model instance that was feasible. If the user sets a resource limit, such as a time limit, an iteration limit, or node limit, for example, or if a user interrupts the process interactively, the conflict that is available at that termination will be the best (that is, the most refined) that was achievable at that point. Even a nonminimal conflict may be more useful than the full model for determining the cause of infeasibility. The status of a bound or constraint in such a nonminimal conflict may be *proved*, meaning that the conflict refiner had sufficient resources to prove participation of bound or constraint in the conflict, or the status may be *possible*, meaning that the conflict refiner has not yet proven whether the bound or constraint is necessarily part of a minimal conflict.

If a model contains more than one cause of infeasibility, then the conflict that is delivered may not be unique. As you saw in the example, you may repair one infeasibility only to find that there is another arising. An iterative approach may be necessary.

When the conflict refiner is allowed to run to completion, a conflict will be minimal in the sense that removal of any constraint or bound will result in a feasible subproblem. However, even if there is a single cause of infeasibility, it is worth realizing that conflicts can often be derived in more than one way, and one minimal conflict may be smaller (fewer in number of

constraints or bounds) than another. For example, consider this small set of inconsistent constraints:

```
x + y + z >= 1
  x          <= 0
    y        <= 0
      z      <= 0
x + y + z <= 0
```

There are multiple minimal conflicts in that small set.

```
(1)
x + y + z >= 1
  x          <= 0
    y        <= 0
      z      <= 0
```

```
(2)
x + y + z >= 1
x + y + z <= 0
```

Removing any one of the constraints in conflict (1) results in feasibility. Likewise, removing either of the constraints in conflict (2) also results in feasibility. Either representation may guide you toward a correct analysis of the infeasibilities in the model.

Keep in mind also that a conflict may guide you toward multiple ways to repair a model, some more reasonable than others. For example, if the conflict in a model using continuous variables to represent percentages looked like this:

```
x1 + x2 + x3 >= 4
Bounds
  0 <= x1 <= 1
  0 <= x2 <= 1
  0 <= x3 <= 1
```

the infeasibility could be repaired by one change, namely, by increasing the upper bound of x_3 to be 2. However, with the way the variables are defined, this modification makes little sense. It is more likely that the model contains two mistaken constraints as shown.

When the model passed to the conflict refiner is actually feasible, the conflict refiner will return this message:

```
Problem is feasible; no conflict available
```

An attempt to display or access a conflict when none exists, whether because the conflict refiner has not yet been invoked or because an error occurred, results in this error message:

```
No conflict exists.
```

The cause of those messages will usually be apparent to a user. However, numeric instability may cause genuine uncertainty for a user. In an unstable model, one of the optimizers may return a valid conclusion of infeasibility, based on the numeric precision allowed by the model, and yet when a trivial modification is made, the model status changes, and a feasible

solution now seems attainable. Because one of the conventional indicators of instability can be this switching back and forth from feasibility to infeasibility, the user should be alert to this possibility. The conflict refiner will halt and return an error code if an infeasible model suddenly appears feasible during its analysis, due to this presumption of numeric instability. The user should turn attention away from infeasibility analysis at that point, and toward the sections in this manual such as *Numeric Difficulties* on page 174.

Using the Conflict Refiner in an Application

Here is an example using the conflict refiner in the C++ API of Concert Technology. You will modify one of the standard examples `ilomipex2.cpp` distributed with the product. Those modifications highlight special considerations about the conflict refiner within an application discussed more fully in *What Belongs in an Application to Refine Conflict* on page 367 and *Conflict Application vs Interactive Optimizer* on page 367.

Starting from that example, locate this statement in it:

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

Immediately after that statement, insert the following lines to prepare for and invoke the conflict refiner:

```

if ( ( cplex.getStatus() == IloAlgorithm::Infeasible ) ||
    ( cplex.getStatus() == IloAlgorithm::InfeasibleOrUnbounded ) ) {
    cout << endl << "No solution - starting Conflict refinement" << endl;

    IloConstraintArray infeas(env);
    IloNumArray preferences(env);

    infeas.add(rng);
    infeas.add(sos1); infeas.add(sos2);
    if ( lazy.getSize() || cuts.getSize() ) {
        cout << "Lazy Constraints and User Cuts ignored" << endl;
    }
    for (IloInt i = 0; i<var.getSize(); i++) {
        if ( var[i].getType() != IloNumVar::Bool ) {
            infeas.add(IloBound(var[i], IloBound::Lower));
            infeas.add(IloBound(var[i], IloBound::Upper));
        }
    }

    for (IloInt i = 0; i<infeas.getSize(); i++) {
        preferences.add(1.0); // user may wish to assign unique preferences
    }

    if ( cplex.refineConflict(infeas, preferences) ) {
        IloCplex::ConflictStatusArray conflict = cplex.getConflict(infeas);
        env.getImpl()->useDetailedDisplay(IloTrue);
        cout << "Conflict :" << endl;
        for (IloInt i = 0; i<infeas.getSize(); i++) {
            if ( conflict[i] == IloCplex::ConflictMember)
                cout << "Proved  : " << infeas[i] << endl;
            if ( conflict[i] == IloCplex::ConflictPossibleMember)
                cout << "Possible: " << infeas[i] << endl;
        }
    }
    else
        cout << "Conflict could not be refined" << endl;
    cout << endl;
}

```

Now run this modified version with the model you have seen in *A Model for the Conflict Refiner* on page 356. You will see results like these:

```
No solution - starting Conflict refinement

Refine conflict on 14 members...

Iteration  Max Members  Min Members
          1             11             0
          2             9             0
          3             5             0
          4             3             0
          5             2             0
          6             2             1
          7             2             2

Conflict :
Proved   : c2( 2.1 <= ( x1 + x2 + 0.8 * x3 + 0.6 * x4 + 0.4 * x5 ) )
Proved   : c8( ( x2 + x3 + x4 + x5 ) <= 0)
```

What Belongs in an Application to Refine Conflict

There are a few remarks to make about that modification:

- ◆ Lazy constraints must not be present in a conflict.
- ◆ User-defined cuts (also known as user cuts) must not be present in a conflict.

These lines check for lazy constraints and user-defined cuts.

```
if ( lazy.getSize() || cuts.getSize() ) {
    cout << "Lazy Constraints and User Cuts ignored" << endl;
}
```

- ◆ Since it makes little sense to modify the bounds of binary (0-1) variables, this example does not include them in a conflict. This line eliminates binary variables from consideration:

```
if ( var[i].getType() != IloNumVar::Bool ) {
```

Eliminating binary variables from consideration produces behavior consistent with behavior of the Interactive Optimizer. Doing so is optional. If you prefer for the conflict refiner to work on the bounds of your binary variables as well, omit this test, bearing in mind that it may take much longer to refine your model to a minimal conflict in that case.

- ◆ The method `useDetailedDisplay` is included to improve readability of the conflict when it is displayed.

Conflict Application vs Interactive Optimizer

This modified example also demonstrates a few features that are available only in the Callable Library and Concert Technology, not in the Interactive Optimizer:

- ◆ *Preferences in the Conflict Refiner* on page 368
- ◆ *Groups in the Conflict Refiner* on page 368

Preferences in the Conflict Refiner

You can assign *preference* to members of a conflict. In most cases there is no advantage to assigning unique preferences, but if you know something about your model that suggests assigning an ordering to certain members, you can do so.

- A preference of -1 means that the member is to be absolutely excluded from the conflict
- A preference of 0 (zero) means that the member is always to be included, and
- Preferences of positive value represent an ordering by which the conflict refiner will give preference to the members. A group with a higher preference is more likely to be included in the conflict. Preferences can thus help guide the refinement process toward a more desirable minimal conflict.

Groups in the Conflict Refiner

You can organize constraints and bounds into one or more *groups* in a conflict. A group is a set of constraints or bounds that must be considered together; that is, if one member of a group is determined by the conflict refiner to be a necessary in a minimal conflict, then the entire group will be part of the conflict.

For example, in the resource allocation problem from *A Model for the Conflict Refiner* on page 356, management might consider the three skill requirements (c2, c3, c4) as inseparable. Adjusting the data in any one of them should require a careful re-evaluation of all three. To achieve that effect in the modified version of `ilomipex2.cpp`, replace this line:

```
infeas.add(rng);
```

by the following lines to declare a group of the constraints expressing skill requirements:

```
infeas.add(rng[0]);
IloAnd skills(env);
skills.add(rng[1]);
skills.add(rng[2]);
skills.add(rng[3]);
infeas.add(skills);
for (IloInt i = 4; i < rng.getSize(); i++) {
    infeas.add(rng[i]);
}
```

(This particular modification is specific to this simplified resource allocation model and thus would not make sense in some other infeasible model you might run with the modified `ilomipex2.cpp` application.)

After that modification, the cost constraint and the constraints indexed 4 through 10 are treated individually (that is, normally) as before. The three constraints indexed 1 through three are combined into a `skills` constraint through the `IloAnd` operator, and added to the infeasible set.

Individual preferences are not assigned to any of these members in this example, but you could assign preferences if they express your knowledge of the problem.

After this modification to group the skill constraints, a minimal conflict is reported like this, with the skill constraints grouped inseparably:

```
Conflict :
Proved   : IloAnd and36 = {
c2( 2.1 <= ( x1 + x2 + 0.8 * x3 + 0.6 * x4 + 0.4 * x5 ) )
c3( 1.2 <= ( x6 + 0.9 * x7 + 0.5 * x8 ) )
c4( 0.8 <= ( x9 + 0.9 * x10 ) ) }

Proved   : c8( ( x2 + x3 + x4 + x5 ) <= 0)
```


Repairing Infeasibilities with FeasOpt

This chapter tells you about FeasOpt, a feature for repairing infeasibility in a model. FeasOpt attempts to repair an infeasibility by modifying the model according to *preferences* set by the user. This chapter covers these topics:

- ◆ *What Is FeasOpt?* on page 371
- ◆ *Invoking FeasOpt* on page 372
- ◆ *Specifying Preferences* on page 373
- ◆ *Example: FeasOpt in Concert Technology* on page 373

What Is 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. FeasOpt supports all types of infeasible models. In essence, FeasOpt is another optimization algorithm (analogous to phase I of the simplex algorithm). It tries to suggest the least change that would achieve feasibility. FeasOpt does not actually modify your model. Instead, it suggests a set of bounds and constraint ranges and produces the solution that would result from these relaxations. Your application can query this solution. It can also 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.

The infeasibility on which FeasOpt works must be present explicitly in your model among its constraints and bounds. In particular, if you have set a MIP cutoff value with the idea that the cutoff value will render your model infeasible, and then you apply FeasOpt, you will not achieve the effect you expect. In such a case, you should add one or more explicit constraints to enforce the restriction you have in mind. In other words, add constraints rather than attempt to enforce a restriction through these parameters:

- `CutLo` or `CutUp` in Concert Technology (**not** recommended to enforce infeasibility)
- `CPX_PARAM_CUTLO` or `CPX_PARAM_CUTUP` in the Callable Library (**not** recommended to enforce infeasibility)
- `mip tolerance lowercutoff` or `uppercutoff` in the Interactive Optimizer (**not** recommended to enforce infeasibility)

Invoking FeasOpt

Depending on the interface you are using, you invoke FeasOpt in one of the ways listed in Table 25.1.

Table 25.1 FeasOpt

API or Component	FeasOpt
Concert Technology for C++ users	<code>IloCplex::feasOpt</code>
Concert Technology for Java users	<code>IloCplex.feasOpt</code>
Concert Technology for .NET users	<code>Cplex.FeasOpt</code>
Callable Library	<code>CPXfeasopt</code> and <code>CPXfeasoptext</code>
Interactive Optimizer	<code>feasopt { variables constraints all }</code>

In the various Concert Technology APIs, you have a choice of three implementations 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, you can allow changes without distinguishing bounds on variables from ranges over constraints.

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.

Specifying Preferences

You specify the bounds or ranges that FeasOpt may consider for modification by assigning positive *preferences* for each. A negative or zero preference means that the associated bound or range is **not** to be modified. One way to construct a weighted penalty function from these preferences is 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.

Example: FeasOpt in Concert Technology

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;
try {
    IloModel model(env);
    IloNumVarArray x(env);
    IloRangeArray con(env);
    IloNumArray vals(env);
    IloNumArray infeas(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.exportModel("toto.lp");
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) ) {
env.out() << endl;
cplex.getInfeasibilities(infeas,con);
env.out() << "*** Suggested bound changes = " << infeas << 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);
}
```

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. 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 permitted.

Then `FeasOpt` is called. If the `FeasOpt` call succeeds, then several lines of output give the results. Here is the output:

```
*** First feasOpt call ***
*** Consider all constraints ***

*** Suggested bound changes = [50, -0, -0]
*** Feasible objective value would be = 50
Solution status      = Infeasible
Solution obj value = 50
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 increasing its lower bound by 50 units.

The solution values of [40, 30, 80] would be feasible in the modified form of the constraint, but not in the original form. 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 righthand 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;

lb[0]=ub[0]=0.0;

if ( cplex.feasOpt(con, lb, ub) ) {
    env.out() << endl;
    cplex.getInfeasibilities(infeas,con);
    env.out() << "*** Suggested bound changes = " << infeas << 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);
}
```

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 bound changes = [-0, -0, -50]
*** Feasible objective value would be = 50
Solution status = Infeasible
Solution obj value = 50
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.) The negative suggested bound change of the third constraint means that `FeasOpt`

suggests decreasing the upper bound of the third constraint by 50 units, transforming this constraint:

$$x[0] + x[1] + x[2] \geq 150$$

into

$$x[0] + x[1] + x[2] \geq 100$$

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 preferences - 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) ) {
  env.out() << endl;
  cplex.getInfeasibilities(infeas,x);
  env.out() << "*** Suggested bound changes = " << infeas << 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 bound changes = [25, 0, 0]
*** Feasible objective value would be = 25
Solution status      = Infeasible
Solution obj value = 25
Values              = [65, 30, 55]
```

Those results suggest modifying only one bound, the upper bound on the first variable. 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;

if ( cplex.feasOpt(x, lb, ub) ) {
  env.out() << endl;
  cplex.getInfeasibilities(infeas,x);
  env.out() << "Suggested bound changes = " << infeas << 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 $[0, +\infty]$, 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.

Part VI

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:

- ◆ *User-Cut and Lazy-Constraint Pools* on page 381
- ◆ *Using Goals* on page 389
- ◆ *Using Callbacks* on page 407
- ◆ *Goals and Callbacks: a Comparison* on page 425
- ◆ *Advanced Presolve Routines* on page 429
- ◆ *Advanced MIP Control Interface* on page 439
- ◆ *Parallel Optimizers* on page 447

User-Cut and Lazy-Constraint Pools

In contrast to the cuts that ILOG CPLEX may automatically add while solving a problem, **user cuts** are those cuts that a user defines based on information already implied about the problem by the constraints; user cuts may not be strictly necessary to the problem, but they tighten the model. **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 constraints. This chapter covers those topics.

- ◆ *What Are Pools of User Cuts or Lazy Constraints?* on page 382
- ◆ *Adding User Cuts and Lazy Constraints* on page 384
- ◆ *Deleting User Cuts and Lazy Constraints* on page 387

Important: Only **linear** constraints may be included in a pool of user cuts or lazy constraints. Neither user cuts nor lazy constraints may contain quadratic terms.

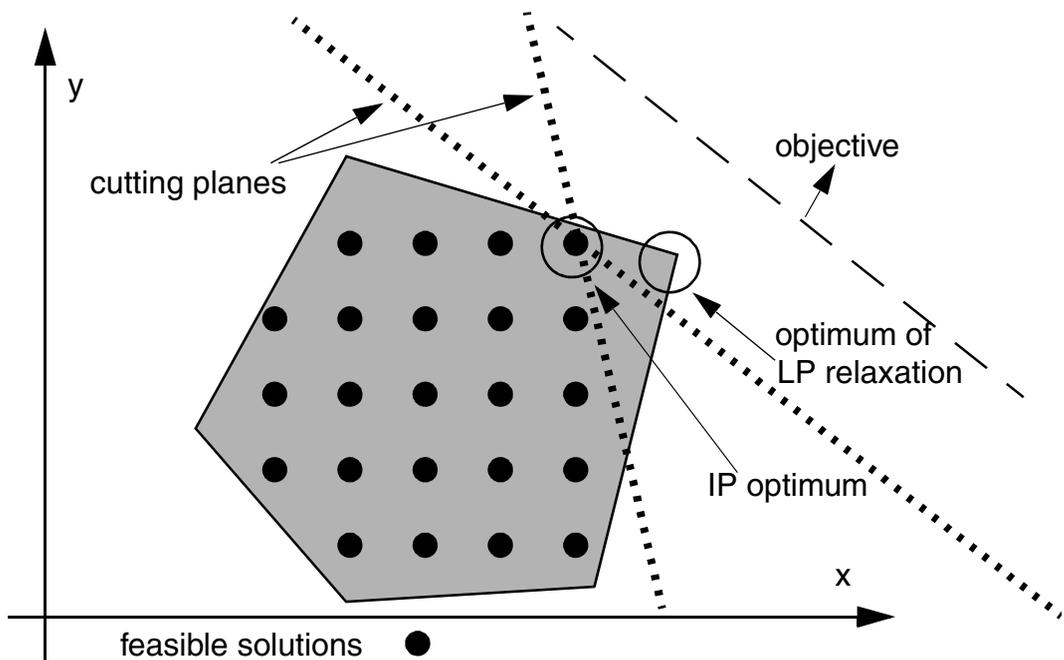


Figure 26.1 Cuts in a typical MIP

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 in one of these ways:

- ◆ through the cut callback described in *Advanced MIP Control Interface* on page 439, or
- ◆ by setting up *cut pools* before MIP optimization begins, as explained in *Adding User Cuts and Lazy Constraints* on page 384.

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 if you are using the Callable Library. (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 routines or methods in the Component Libraries or via LP, SAV, or MPS files, as explained in the following sections.

- ◆ *Using Component Libraries* on page 384
- ◆ *Using the Interactive Optimizer* on page 384
- ◆ *Reading and Writing LP Files* on page 384
- ◆ *Reading and Writing SAV Files* on page 386
- ◆ *Reading and Writing MPS Files* on page 386

Using Component Libraries

The following facilities will add user defined cuts to a user cut pool.

- ◆ The CPLEX Callable Library routine `CPXaddusercuts`
- ◆ The Concert Technology methods:
 - `IloCplex::addUserCuts` in the C++ API
 - `IloCplex.addUserCuts` in the Java API
 - `Cplex.AddUserCuts` in the .NET API

The following facilities will add lazy constraints to a lazy constraint pool.

- ◆ The CPLEX Callable Library routine is `CPXaddlazyconstraints`.
- ◆ The Concert Technology methods
 - `IloCplex::addLazyConstraints` in the C++ API
 - `IloCplex.addLazyConstraints` in the Java API
 - `Cplex.AddLazyConstraints` in the .NET API

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.

Reading and Writing LP Files

User cuts and lazy constraints may also be specified in LP-format files, and so may be read:

- ◆ With the Interactive Optimizer `read` command

- ◆ Through the routine `CPXreadcopyprob` of the Callable Library
- ◆ Through the methods of Concert Technology:
 - `IloCplex::importModel` of the C++ API
 - `IloCplex.importModel` of the Java API
 - `Cplex.ImportModel` of the .NET API

When CPLEX writes LP-format files, user cuts and lazy constraints added through their respective `add` routines or read from LP format files will be included in the output files along with their names (if any).

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 an LP 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.

Reading and Writing SAV Files

User cuts and lazy constraints may also be specified SAV-format files, and so may be read:

- ◆ With the Interactive Optimizer `read` command
- ◆ Through the routine `CPXreadcopyprob` of the Callable Library
- ◆ Through the methods of Concert Technology:
 - `IloCplex::importModel` of the C++ API
 - `IloCplex.importModel` of the Java API
 - `Cplex.ImportModel` of the .NET API

When CPLEX writes SAV format files, user cuts and lazy constraints added through their respective `add` routines or read from SAV format files will be included in the output files along with their names (if any).

Reading and Writing MPS Files

ILOG CPLEX extends the MPS file format with additional optional sections to accommodate user defined cuts and lazy constraints. The usual routines of the Callable Library and methods of Concert Technology to read and write MPS files also read and write these optional sections. These additional sections follow the `ROWS` section of an MPS file in this order:

- ◆ `ROWS`
- ◆ `USERCUTS`
- ◆ `LAZYCONS`

The syntax of these additional sections conforms to the syntax of the `ROWS` section with this exception: the type `R` cannot appear in `USERCUTS` nor in `LAZYCONS`. For details about the format of the `ROWS` section in the MPS file format, see the *ILOG CPLEX File Format Reference Manual*, especially these sections:

- ◆ *ROWS Section* on page 16
- ◆ *User Defined Cuts in MPS Files* on page 28
- ◆ *Lazy Constraints in MPS Files* on page 29

Here is an example of an MPS file extended to include lazy constraints.

```

NAME          extra.mps
ROWS
  N  obj
  L  c2
  L  c3
LAZYCONS
  L  c1
COLUMNS
  MARK0000  'MARKER'          'INTORG'
  x1      obj          -12
  x1      c2           2
  x1      c3           3
  x1      c1           5
  x2      obj          -5
  x2      c2           3
  x2      c3           2
  x2      c1           1
  x3      obj          -15
  x3      c2           4
  x3      c3           4
  x3      c1           9
  x4      obj          -10
  x4      c2           1
  x4      c3          10
  x4      c1          12
  MARK0001  'MARKER'          'INTEND'
RHS
  rhs      c2          10
  rhs      c3           8
  rhs      c1          15
BOUNDS
  UP bnd   x1           5
  UP bnd   x2           5
  UP bnd   x3           5
  UP bnd   x4           5
ENDATA

```

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.

Using Goals

This chapter explores goals and their role in a branch & cut search. In it, you will learn about:

- ◆ *Branch & Cut with Goals* on page 390
- ◆ *Special Goals in Branch & Cut* on page 392
- ◆ *Aggregating Goals* on page 394
- ◆ *Example: Goals in Branch & Cut* on page 395
- ◆ *The Goal Stack* on page 397
- ◆ *Memory Management and Goals* on page 398
- ◆ *Cuts and Goals* on page 399
- ◆ *Injecting Heuristic Solutions* on page 401
- ◆ *Controlling Goal-Defined Search* on page 402
- ◆ *Example: Using Node Evaluators in a Node Selection Strategy* on page 404
- ◆ *Search Limits* on page 406

Note: Goals are implemented by `IloCplex::Goal`, not `IloGoal` as in other ILOG products.

Branch & Cut with Goals

Goals allow you to take control of the branch & cut search procedure used by ILOG CPLEX to solve MIP problems. To help you understand how to use goals with ILOG CPLEX, these sections review how this procedure works.

- ◆ *Overview of Goals in Branch & Cut* on page 390
- ◆ *How Goals Are Implemented in Branch & Cut* on page 391
- ◆ *About the Method execute in Branch & Cut* on page 391

Overview of Goals in Branch & Cut

The 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. ILOG CPLEX 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, ILOG CPLEX starts by solving the continuous relaxation of its subproblem (that is, the subproblem without integrality constraints). If the solution violates any cuts, ILOG CPLEX adds them to the node problem and re-solves. This procedure is iterated until no more violated cuts are found by ILOG CPLEX. 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, ILOG CPLEX 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, ILOG CPLEX splits the node problem into one or two smaller subproblems, typically by branching on a variable that violates its integrality constraint. 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.

How Goals Are Implemented in Branch & Cut

Note: The discussion of the details of using goals will be presented mainly in terms of the C++ API. The Java and .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). In .NET, goals are implemented by the class `Cplex.Goal`. The method `IloCplex::GoalI::execute` (`IloCplex.Goal.execute`) is where the control is implemented. This method is called by `IloCplex` after a node relaxation has been solved and all cuts have been added. Invoking the 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` or `Cplex.Goal`) can be called to query information about the current node problem and the solution of its relaxation.

About the Method `execute` in Branch & Cut

Typically, the implementation of the 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. Note, however, that this CP-style application of goals prevents you from making use of MP-style advanced starting information, either from a MIP Start file or by restarting a previous optimization with one that uses goals.
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 ILOG CPLEX.
4. Instruct ILOG CPLEX how to proceed. Instruct ILOG CPLEX how to proceed through the return value of the method `execute`; the return value is `execute` is another goal. ILOG CPLEX simply continues by executing this goal.

Special Goals in Branch & Cut

ILOG CPLEX provides a selection of special goals that can be used to specify how to proceed in branch & cut. These special goals are:

- ◆ *Or Goal* on page 392
- ◆ *And Goal* on page 392
- ◆ *Fail Goal* on page 392
- ◆ *Local Cut Goal* on page 393
- ◆ *Null Goal* on page 393
- ◆ *Branch as CPLEX Goal* on page 393
- ◆ *Solution Goal* on page 394

Or Goal

An Or goal is a goal that creates subnodes of the current node. This function takes at least 2 and up to 6 goals as arguments. For each of its arguments, the Or goal will create a subnode in such a way that when that subnode is processed, the corresponding goal will be executed. After the goal has been executed, the current node is immediately deactivated.

In the C++ API, an Or goal is returned by `IloCplex::GoalI::OrGoal`.

In the Java API, an Or goal is returned by the method `IloCplex.or`.

In the .NET API, an Or goal is returned by the method `Cplex.Or`.

And Goal

An And goal also takes goals as arguments. It returns a goal that will cause ILOG CPLEX to execute all the goals passed as arguments in the order of the arguments.

In the C++ API, an And goal is returned by `IloCplex::GoalI::AndGoal`.

In the Java API, an And goal is returned by the method `IloCplex.and`.

In the .NET API, an And goal is returned by the method `Cplex.And`.

Fail Goal

A Fail goal creates a goal that causes ILOG CPLEX to prune the current node. In other words, it discontinues the search at the node where the goal is executed. ILOG CPLEX will continue with another active node from the tree, if one is available.

In the C++ API, a Fail goal is returned by `IloCplex::GoalI::FailGoal`

In the Java API, a Fail goal is returned by the method `IloCplex.failGoal`.

In the .NET API, a Fail goal is returned by the method `Cplex.FailGoal`.

Local Cut Goal

A *local cut goal* adds a local cut to the node where the goal is executed.

In the C++ API, the class `IloCplex::Goal` has constructors that take an instance of `IloRange` or an instance of `IloRangeArray (IloRange[])` as an argument. When one of these constructors is used, a local cut goal is created.

To create local cut goals with the Java API, use the method `IloCplex.constraintGoal` or if more convenient, one of the methods `IloCplex.leGoal`, `IloCplex.geGoal` or `IloCplex.eqGoal`.

In the .NET API, use the methods `Cplex.ConstraintGoal`, `Cplex.EqGoal`, `Cplex.LeGoal`, or `Cplex.GeGoal` to create a local cut goal.

Null Goal

The 0-goal is also known as a null goal or empty goal.

In the C++ API, a null goal is an `IloCplex::Goal` handle object with a null (0) implementation pointer.

A null goal can also be returned by the method `IloCplex::GoalI::execute`.

Use a null goal when you want to instruct ILOG CPLEX not to branch any further. For example, when CPLEX finds a feasible solution, and you want to accept it without further branching, a null goal is appropriate.

For example, the following sample from the C++ API accepts an integer feasible solution and stops branching.

```
if ( isIntegerFeasible() )
    return 0;
```

Branch as CPLEX Goal

Instead of instructing ILOG CPLEX not to branch any further (as you can do with a null goal), it is also possible to tell ILOG CPLEX to branch as it normally would; in other words, to branch as CPLEX. You give ILOG CPLEX this instruction by means of a branching goal. A branching goal tells ILOG CPLEX to take over control of the branch & cut search with its built-in strategies.

The following lines in the C++ API tell ILOG CPLEX to continue branching as it would normally do:

```
if ( !isIntegerFeasible() )
    AndGoal(BranchAsCplexGoal(getEnv()), this);
```

Solution Goal

A solution goal is a goal that injects a solution, such as a solution supplied by you or a solution found by ILOG CPLEX during the search.

For example, here is a sample in the C++ API which injects a solution at the beginning of branch & bound and then uses a branching goal:

```
if (getNnodes() == 0)
    goal = AndGoal(OrGoal(SolutionGoal(goalvar, startVals),
                          AndGoal(MyBranchGoal(getEnv(), var), goal)),
                  goal);
else
    goal = AndGoal(MyBranchGoal(getEnv(), var), goal);
return goal;
```

For more about this kind of special goal, see *Injecting Heuristic Solutions* on page 401.

Aggregating Goals

Since And goals and Or goals take other goals as arguments, 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, it looks like this:

```
return cplex.and(cplex.or(cplex.leGoal(var, Math.floor(val)),
                          cplex.geGoal(var, Math.floor(val)+1)), this);
```

and for C#.NET, it looks like this:

```
return cplex.And(
    cplex.Or(cplex.GeGoal(_vars[bestj], System.Math.Floor(x[bestj])+1),
             cplex.LeGoal(_vars[bestj], System.Math.Floor(x[bestj]))),
    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 name `IloCplex::GoalI::OrGoal` can be abbreviated to `OrGoal`. Likewise, the full name `IloCplex::GoalI::AndGoal` can be abbreviated to `AndGoal`.

This return statement returns an `And` goal that first executes the `Or` goal and then the current goal itself specified by the `this` argument. When the `Or` goal is executed next, it will create two subnodes. In the first subnode, the first local cut goal representing `var ≤ [val]` (where `[val]` denotes the floor of `val`) will be executed, thus adding the constraint `var ≤ [val]` for the subtree of this node. Similarly, the second subnode will be created, and when executing its constraint goal the constraint `var ≥ [val] + 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 about how goals are processed are available in *The Goal Stack* on page 397, *Controlling Goal-Defined Search* on page 402, and *Search Limits* on page 406.

Example: Goals in Branch & Cut

Consider the following example to clarify the discussions of goals. This example is available as `ilogoalex1.cpp` in the `examples/src` subdirectory of your ILOG CPLEX distribution. The equivalent Java implementation can be found as `GoalEx1.java` in the same location. The C#.NET version is in `GoalEx1.cs` and the VB.NET version is in `GoalEx1.vb`.

This example shows how to implement and use a goal for controlling the branch strategy used by ILOG CPLEX. As discussed, goals are implemented as subclasses of the class `IloCplex::GoalI (IloCplex.Goal or Cplex.Goal)`. The C++ implementation 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 or Cplex.Goal)` and has a private member `IloNumVarArray vars`. The function `MyBranchGoal` creates an instance of class `MyBranchGoalI`, initializes the member `vars` to the argument `vars` passed to the function, and returns a handle to the new goal object. The curly brackets "`{ ... }`" following the macro enclose the implementation of the 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 seven 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 or Cplex.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`, starts by declaring and initializing some arrays. These arrays are then used by methods of class `IloCplex::GoalI(IloCplex.Goal or Cplex.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 or Cplex.Goal)` provides a wealth of other query methods. For details, see the *ILOG CPLEX Reference Manuals*.

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 `vars[bestj] ≤ ⌊x[bestj]⌋` and one for `vars[bestj] ≥ ⌊x[bestj]⌋ + 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 the 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 understand how goals are executed, consider the concept of the *goal stack*. Every node has its own goal stack. When `cplex.solve(goal)` is called, the goal stack of the root node is simply initialized with `goal` and then the regular `cplex.solve` method is called.

When ILOG CPLEX 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. ILOG CPLEX keeps doing this until the node becomes inactive or the node's goal stack becomes empty. When the node stack is empty, ILOG CPLEX 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:

- ◆ As explained in *Or Goal* on page 392, the `Or` goal creates child nodes. ILOG CPLEX 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 argument to the `Or` goal on the goal stack of the corresponding node. Finally, the current node is deactivated, and ILOG CPLEX continues search by picking a new active node from the tree to process.
- ◆ The `And` goal simply pushes the goals passed as arguments onto the goal stack in reverse order. Thus, when the goals are popped from the stack for execution, they will be executed in the same order as they were passed as arguments to the `And` goal.
- ◆ When a `Fail` goal executes, the current node is simply deactivated, and ILOG CPLEX continues on another active node from the tree. In other words, ILOG CPLEX discontinues its search below the current node.
- ◆ When a local cut goal is executed, its constraints are added to the node problem as local cuts and the relaxation is re-solved.
- ◆ An empty goal cannot be executed. Thus, empty goals are not pushed onto the goal stack. If the goal stack is empty, ILOG CPLEX 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 ILOG CPLEX 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 $\text{var} \leq \lfloor \text{val} \rfloor$ (where $\lfloor \text{val} \rfloor$ denotes the floor of `val`) on the goal stack of the first subnode. Similarly, it pushes a local cut goal for $\text{var} \geq \lfloor \text{val} \rfloor + 1$ on the goal stack of the second subnode. Finally, the current node is deactivated and ILOG CPLEX continues its search with a new active node from the tree.

When ILOG CPLEX 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 ILOG CPLEX 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 and .NET use garbage collection to handle all memory management issues. Thus the following applies only to the C++ library. Java or .NET users may safely skip ahead to *Cuts and Goals* on page 399.

To conserve memory, in the C++ API, ILOG CPLEX only stores active nodes of the tree and deletes nodes as soon as they become inactive. When deleting nodes, ILOG CPLEX also deletes the goal stacks associated with them, including all goals they may still contain. In other words, ILOG CPLEX 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.

ILOG CPLEX 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 the C++ API of ILOG Concert Technology. Instead, ILOG CPLEX 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 arguments of the `Or` goal. ILOG CPLEX 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 arguments 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 arguments to a new goal.

Also, user code in the `execute` method is not allowed to modify existing ILOG Concert Technology objects in any way. ILOG CPLEX 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.

Just as you can add local cuts by means of a local cut goal, as explained in *Local Cut Goal* on page 393, you can add a global cut by means of a global cut goal. A global cut goal is created with the method `IloCplex::GoalI::GlobalCutGoal` (`IloCplex.globalCutGoal` or `Cplex.GlobalCutGoal`). This method takes an instance of `IloRange` or `IloRangeArray(IloRange[])` as its argument 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 of only 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 the problem has been interpreted 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 `noswt` model with these cuts is simply to add them to the model before calling the optimizer. However, for demonstration purposes here, 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 implementation appears as `GoalEx2.java` in the same location. Likewise, there is also the C#.NET version in `GoalEx2.cs` and the VB.NET version in `GoalEx2.vb`.

The goal `CutGoal` in that 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 ILOG CPLEX would use on its own.

The goal starts out by checking whether the solution of the continuous relaxation of the current node subproblem is integer feasible. This check is done by the 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 ILOG CPLEX to continue on its own.

Otherwise, the goal checks whether 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 righthand side 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 the method `GlobalCutGoal`. It is then combined with the goal named `goal` by the method `AndGoal`, so that the new global cut goal will be executed first. The resulting goal is stored again in `goal`. Before adding any global cut goals, the `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 ILOG CPLEX 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 the method `IloCplex::GoalI::SolutionGoal(IloCplex.solutionGoal` or `Cplex.SolutionGoal)`. Such a goal can be returned typically as a subgoal of an `And` goal much like global cut goals.

When ILOG CPLEX 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 whether an injected solution is feasible, ILOG CPLEX 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, ILOG CPLEX makes sure that either the solution is feasible with respect to all goals or otherwise it is 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, ILOG CPLEX 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 remains 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 ILOG CPLEX 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 ILOG CPLEX 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 ILOG CPLEX chooses when it selects the next node for processing. ILOG CPLEX has several built-in node selection strategies, selected through the parameter `NodeSel` (`CPX_PARAM_NODESEL`).

When you use goal-controlled search, you use node evaluators to override the built-in node selection strategy. You combine a goal with a node evaluator by calling the method `IloCplex::Goal::Apply` (`IloCplex.apply` or `Cplex.Apply`). This method returns a new goal that implements the same search strategy as the goal passed as the argument, but adds the node evaluator to every node in the subtree defined by the goal. Consequently, nodes may have a list of evaluators attached to them.

When node evaluators are used, nodes are selected like this:

1. ILOG CPLEX starts to choose the node with the built-in strategy as a first candidate.
2. Then ILOG CPLEX loops over all remaining active nodes and considers choosing them instead.
3. If a node has the same evaluator attached to it as the current candidate, the evaluator is asked whether 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. The application order is determined like this:

- 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 evaluations.

In the C++ API, node evaluators are implemented as subclasses of class `IloCplex::NodeEvaluatorI`. The 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 the method `evaluate` in C++:

```
IloNum IloCplex::NodeEvaluatorI::evaluate();
```

and in Java, by the method:

```
double IloCplex.NodeEvaluator.evaluate();
```

and in C#.NET:

```
double Cplex.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` or `Cplex.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 comparison happens only 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. You can alter his behavior 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);
```

or, in the case of C#.NET:

```
bool Cplex.NodeEvaluator.Subsume(double evalNode, double evalCurrent);
```

ILOG CPLEX calls this method of an evaluator attached to the current candidate if the node being compared also has the same evaluator attached. The first argument `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 arguments. 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 you implement 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 the methods of the class `IloCplex::NodeEvaluatorI` (`IloCplex.NodeEvaluator`).

Finally, in C++, the 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: Using Node Evaluators in a Node Selection Strategy

The example `ilogoalex3.cpp` shows how to use node evaluators to implement a node selection strategy that chooses the deepest active node in the tree among those nodes with a maximal sum of integer infeasibilities. The example `ilogoalex3.cpp` can be found in the `examples/src` directory of your distribution. The equivalent Java implementation can be found in the file `Goalex3.java` at the same location. Likewise, the C#.NET example is available in `Goalex3.cs`.

As this example is an extension of the example `ilogoalex1.cpp`, this exposition of it concentrates only on their differences. Also, the example is discussed only in terms of the C++ implementation; the Java implementation has identical structure and design and differs only in syntax, as does the .NET as well.

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 ILOG CPLEX 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 ILOG CPLEX 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 ILOG CPLEX branch strategy, you can use the goal returned by the method

`IloCplex::GoalI::BranchAsCplexGoal(IloCplex.branchAsCplex)`. A goal that

follows the branching performed by the built-in strategy of `IloCplex` 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` or `Cplex.SearchLimit`), and the procedure for implementing and using them is very similar to that for node evaluators. See the reference manuals for more details about implementing and using search limits.

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 439. This chapter includes information about:

- ◆ *Diagnostic Callbacks* on page 408
- ◆ *Implementing Callbacks in ILOG CPLEX with Concert Technology* on page 408
- ◆ *Example: Deriving the Simplex Callback `ilolpex4.cpp`* on page 412
- ◆ *Implementing Callbacks in the Callable Library* on page 414
- ◆ *Interaction Between Callbacks and ILOG CPLEX Parallel Optimizers* on page 416
- ◆ *Example: Using Callbacks `lpex4.c`* on page 416
- ◆ *Control Callbacks for `IloCplex`* on page 417
- ◆ *Example: Controlling Cuts `iloadmipex5.cpp`* on page 418

Notes: The callback class hierarchy for Java and .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 .NET correspond very closely to those in the Java API. However, the name of a .NET class does not begin with `Ilo`. Furthermore, the names of .NET methods are capitalized (that is, they begin with an uppercase character) according to .NET conventions.

For example, the corresponding callback class in .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 algorithm.
- ◆ 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++ implementation of `IloCplex`. It points to an implementation object of a subclass of `IloCplex::CallbackI`. In Java and .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 .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` in Java and `Cplex.Callback` in .NET) 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 Java and .NET do not provide macros, only the more complex way is available for Java or .NET users. This section first explains the more complex way and discusses the underlying design. To implement your C or C++ callback quickly without details about the internal design, proceed directly to *Writing Callbacks with Macros* on page 410.

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 nor for .NET, but `new` is called explicitly for creating a callback object when needed. After an implementation object of your callback is created (either with the constructor function in C++ or by directly calling the `new` operator for Java or .NET), use it with `IloCplex` by calling `cplex.use` with the callback object as an argument. 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 or .NET, there is no way of removing individual callbacks from your `IloCplex` or `Cplex` object. Instead, you

can remove all callbacks by calling `cplex.clearCallbacks`. Since Java and .NET use garbage collection for memory management, there is nothing equivalent to the `end` method for callbacks in the Java or .NET 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 arguments.

Writing Callbacks with Macros

This is how to implement a callback using macros. Since macros are not supported in Java nor in .NET, 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 argument, the macro is `ILOSIMPLEXCALLBACK1`. Generally, for every callback type `XXX` and any number of arguments `n` from 0 to 7, there is a macro called `ILOXXXCALLBACKn`. Table 28.1 lists the callbacks and the corresponding macros and classes (where `n` is a placeholder for 0 to 7).

Table 28.1 *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 argument:

```
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 the variable `mycallback`. This allows you to call `mycallback.endto` to remove the callback from `cplex`. If you do not intend to access your callback (for example, in order to delete it before ending the environment), you may safely leave out the declaration and initialization of the 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 ended 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 arguments with the code enclosed in the outer `{}`. In Java, the same callback is defined like this:

```
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 arguments are passed to the constructor of the callback. For callbacks requiring up to 7 arguments, similar macros are defined where the 0 is replaced by the number of arguments, ranging from 1 through 7. For an example using the cut callback, see *Example: Controlling Cuts iloadmipex5.cpp* on page 418. If you need more than 7 arguments, 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:

```
cpplex.use(MyCallback(env));
```

in C++ or

```
plex.use(new MyCallback());
```

in Java, or in .NET

```
plex.Use(new MyCallback());
```

In the case of C++, function `MyCallback` creates an instance of the implementation class `MyCallbackI`. A handle to this implementation object is passed to `plex` 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, `ilolplex4.cpp`, appears online in the standard distribution at [yourCPLEXinstallation/examples/src](#).

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. After 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 Callable Library Reference Manual*. In that reference manual, the group `optim.cplex.callable.callbacks` gives you direct access to callback routines.

Setting Callbacks

In the Callable Library, diagnostic callbacks are organized into two groups: LP callbacks (that is, continuous callbacks) and MIP callbacks (that is, discrete callbacks). For each group, one callback function can be set by the routine `CPXsetlpcallbackfunc` and one by `CPXsetmipcallbackfunc`. You can distinguish between the actual callbacks by querying the argument `wherfrom` passed to the callback function as an argument by ILOG CPLEX.

The continuous callback is also called during the solution of problems of type LP, QP, and QCP.

Callbacks for Continuous and Discrete Problems

ILOG CPLEX will evaluate two user-defined callback functions, one during the solution of continuous problems and one during the solution of discrete problems. ILOG CPLEX calls the continuous callback once per iteration during the solution of an LP, QP, or QCP problem and periodically during the presolve. ILOG CPLEX calls the discrete 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 28.2.

Table 28.2 *Status of Nonzero Callbacks for MIPs*

Value	Symbolic constant	Meaning
113	<code>CPXMIP_ABORT_FEAS</code>	current solution integer feasible
114	<code>CPXMIP_ABORT_INFESAS</code>	no integer feasible solution found

Interaction Between Callbacks and ILOG CPLEX Parallel Optimizers

When you use callback routines, and invoke the parallel implementation 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 28.3 summarizes those status variables.

Table 28.3 *Status Variables in `lpex4.c`*

Variable	Represents status returned by this routine
<code>frstatus</code>	<code>CPXfreeprob</code>
<code>clstatus</code>	<code>CPXcloseCPLEX</code>

- ◆ 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 online in the standard distribution at *yourCPLEXinstallation/examples/src*.

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 408 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 and for .NET 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 denoted by `IloCplex.BranchCallback`. Likewise, in .NET, you will find the corresponding class `Cplex.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, expressed as pseudo C++, look like this:

```
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 derive from an interpretation of the model as a resource allocation problem 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. In case you want to copy and paste those cuts into a model in the Interactive Optimizer, for example, here are the same cuts expressed in the conventions of the Interactive Optimizer with uppercase variable names, as in the MPS data file:

```
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
```

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 an argument and adds individual cuts whenever they are violated by 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++ implementation of the callback. In C++, the callback is implemented with these lines:

```
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 arguments, 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 with ellipsis (...) representing the actual implementation:

```
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() {
    ...
}
```

Similar macros are provided for other numbers of arguments ranging from 0 through 7 for all callback classes.

The first argument `lhs` is an array of expressions, and the argument `rhs` is an array of values. These arguments are the lefthand side and righthand side values of cuts of the form $lhs \leq rhs$ to be tested for violation and potentially added. The third argument `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 an argument. This value is tested for violation of more than the tolerance argument `eps` with the corresponding righthand side value.

Tip: A numeric 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 numeric 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 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 `noswt` model which are passed to this function as the argument `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 righthand 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 unnecessarily evaluating it again.

As for the C++ implementation, 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 `noswt` 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 online in the standard distribution at [yourCPLEXinstallation/examples/src](#). The Java version is found in file `AdMIPex5.java` at the same location. The C#.NET implementation is in `AdMIPex5.cs` and the VB.NET implementation is in `AdMIPex5.vb`.

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.
- ◆ Supporting advanced starts
 - Since goals can enforce constraints, they do **not** support advanced starting information. An optimization with goals starts from scratch.
 - Since each callback provides a specific functionality, callbacks support advanced starts.

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 modeled 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.

Advanced Presolve Routines

This chapter explains how to use the advanced presolve routines. These advanced routines are available only in the Callable Library (C API). The topics are:

- ◆ *Introduction to Presolve* on page 430
- ◆ *Restricting Presolve Reductions* on page 432
- ◆ *Manual Control of Presolve* on page 434
- ◆ *Modifying a Problem* on page 437

Introduction to Presolve

This discussion of the advanced presolve interface begins 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 everyone is encouraged 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 unresolves 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 at the beginning of the optimization and applied a second time to the root relaxation, unless the relaxation preprocessing indicator `RelaxPreInd` or `CPX_PARAM_RELAXPREIND` is set to 0 (zero), in which case the presolve is performed only

once. 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 432.) 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 430, 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. The following sections document special considerations about presolve reductions.

- ◆ *Adding Constraints to the First Solution* on page 432
- ◆ *Primal and Dual Considerations in Presolve Reductions* on page 433
- ◆ *Cuts and Presolve Reductions* on page 433
- ◆ *Infeasibility or Unboundedness in Presolve Reductions* on page 434
- ◆ *Protected Variables in Presolve Reductions* on page 434

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 $-x_1 + x_2 + x_3$	min $6y_1 + 5y_2$
st $x_1 + x_2 + 2x_3 \leq 6$	st $y_1 \geq -1$
$x_2 + x_3 \leq 5$	$y_1 + y_2 \geq 1$
0	$2y_1 + y_2 \geq 1$
$x_1, x_2, x_3 \geq 0$	$y_1, y_2, y_3 \geq 0$

Note that the first constraint in the dual ($y_1 \geq -1$) is redundant. Presolve can use this information about the dual problem (and complementary slackness) to conclude that variable x_1 can be fixed to 0 and removed from the presolved problem. While it may be intuitively obvious from inspection of the primal problem that x_1 can be fixed to 0, it is important to note that dual information (redundancy of the first dual constraint) is used to prove it formally.

Now consider the addition of a new constraint $x_2 \leq 5x_1$:

Primal:	Dual:
max $-x_1 + x_2 + x_3$	min $6y_1 + 5y_2$
st $x_1 + x_2 + 2x_3 \leq 6$	st $y_1 - 5y_3 \geq -1$
$x_2 + x_3 \leq 5$	$y_1 + y_2 + y_3 \geq 1$

$$\begin{array}{rcll}
 -5x_1 + x_2 & \leq & 0 & 2y_1 + y_2 \geq 1 \\
 x_1, x_2, x_3 & \geq & 0 & y_1, y_2, y_3 \geq 0
 \end{array}$$

Our goal is to add the appropriate constraint to the presolved problem and reoptimize. Note, however, that the dual information presolve used to fix x_1 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 $x_1=1$, $x_2=5$, $x_3=0$). As a result, CPLEX is unable to use the presolved problem to reoptimize.

Primal and Dual Considerations in Presolve Reductions

Presolve reductions can be classified into several groups: those that rely on primal information, those that rely on dual information, and those that rely on both. Addition of new constraints, modifications to objective coefficients, and tightening of variable bounds (a special case of adding new constraints) require the user to turn off dual reductions. Introduction of new columns, modifications to righthand-side values, and relaxation of variable bounds (a special case of modifying righthand-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, ILOG 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.

Cuts and Presolve Reductions

Cutting planes in mixed integer programming are handled somewhat differently than one might expect. If a user wishes to add his or her own cuts during the branch & cut process (through `CPXaddusercuts` or `CPXcutcallbackadd`), it may appear necessary to turn off dual reductions to accommodate them. (In fact, in this respect, these cuts differ from lazy constraints discussed in *User-Cut and Lazy-Constraint Pools* on page 381.) 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 (zero) to forbid these latter reductions.

Infeasibility or Unboundedness in Presolve 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.*

Protected Variables in Presolve Reductions

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 parameter `CPX_PARAM_REDUCE` to restrict the types of presolve reductions, CPLEX can make use of the optimal basis of 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`—`CPXqpuncrushpi` for QPs—to unpresolve solutions from the presolved problem, creating solutions for the original problem.

The routine `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 call to `CPXcrushform` 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. `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 on the problem. Recall that if primal reductions are turned off, the user can add variables, change the righthand-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.

Advanced MIP Control Interface

In this manual, *Using Callbacks* on page 407 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 440
- ◆ *Heuristic Callback* on page 441
- ◆ *Cut Callback* on page 442
- ◆ *Branch Selection Callback* on page 443
- ◆ *Incumbent Callback* on page 444
- ◆ *Node Selection Callback* on page 445
- ◆ *Solve Callback* on page 445

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 277 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. Consequently, it's a good idea to become familiar with *Advanced Presolve Routines* on page 429, before you read this chapter.

Control callbacks in the ILOG Concert Technology CPLEX Library use the variables of the original model. 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 (`CPXgetcallbackglobalb`, 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`, `CPXgetcallbackglobalb`, 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 381.

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. Depending on the API, the callback function sets a parameter or invokes a method to indicate whether or not the new solution should replace the incumbent solution.

For the object-oriented callback classes of the C++, Java, and .NET APIs, all callback information about the model and solution vector pertains to the original, unpresolved model. For the C API, 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.*

Parallel Optimizers

This chapter tells you how to use ILOG CPLEX parallel optimizers: Parallel Barrier, Parallel MIP, and Concurrent. These parallel optimizers are available as a separate product from ILOG CPLEX. They are implemented to run on hardware platforms with parallel processors. These parallel optimizers, though separate products, can be called from the Interactive Optimizer and the Component Libraries.

In this chapter, you will learn about:

- ◆ *Threads* on page 448
- ◆ *Nondeterminism* on page 449
- ◆ *Clock Settings and Time Measurement* on page 450
- ◆ *Using Parallel Optimizers in the Interactive Optimizer* on page 450
- ◆ *Using Parallel Optimizers in the ILOG CPLEX Component Libraries* on page 451
- ◆ *Parallel Barrier Optimizer* on page 451
- ◆ *Concurrent Optimizer* on page 452
- ◆ *Parallel MIP Optimizer* on page 452

Threads

The ILOG CPLEX parallel optimizers are licensed for a specific maximum number of *threads*. 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 32.1 summarizes the values of those thread-limit parameters.

Table 32.1 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.

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 processor as its threads compete with one another.

Threads and Performance Considerations

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 numeric 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 448.
3. Enter your problem object and populate it as usual.

4. Call the parallel optimizer with the appropriate command from Table 32.2.

Table 32.2 *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 109 and *Instantiate the Problem Object* on page 110 for details.
2. Within your application, set the appropriate ILOG CPLEX parameter from Table 32.1 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 110.
4. Call the parallel optimizer with the appropriate method or routine from Table 32.3.

Table 32.3 *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 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 *Solving LPs: Barrier Optimizer* on page 187, and explored further in *Solving Problems with a Quadratic Objective (QP)* on page 217 and in *Solving Problems with*

Quadratic Constraints (QCP) on page 229.) 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 454.

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.

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 453
- *Branch & Cut Parallel Processing* on page 453
- *Root Relaxation Parallel Processing* on page 453
- *Individual Optimizer Parallel Processing* on page 454
- *Nested Parallel Processing* on page 454

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 454
- ◆ *Output from the Parallel MIP Optimizer* on page 454

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 or QP 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 barrier optimizer alone is used.

You can try a different form of parallelism at the root by selecting the barrier optimizer specifically with the starting algorithm parameter (`RootAlg` in Concert, `CPX_PARAM_STARTALG` in the Callable Library, `set mip strategy startalgorithm` in the Interactive 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, setting the parameter `NodeAlg` to `barrier`, 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 presolved model.

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 273) 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
      Node Left      Objective  IInf Best Integer      Cuts/
                                     Best Node      ItCnt      Gap
*      0      0      3192.0420      12
      0+      0      3384.5860      15      4505.0000      3192.0420      8      29.14%
                                     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      3604.4590      17      4471.0000      3530.1967      104      21.04%
      3607.9420      18      4471.0000 Flowcuts: 10      124      19.38%
*      0+      0      3608.7548      20      4448.0000      3607.9420      131      19.30%
      3617.6257      19      4448.0000      3607.9420      131      18.89%
      3624.7454      19      4448.0000 Flowcuts: 3      136      18.87%
      3627.2428      20      4448.0000 Flowcuts: 2      141      18.67%
                                     Flowcuts: 2      150      18.51%
*      0+      0      3994.0000      0      4448.0000 Flowcuts: 1      152      18.45%
*      30+      5      3985.0000      0      3994.0000      3627.2428      152      9.18%
*      46      0      3983.0000      0      3985.0000      3736.8034      305      6.23%
                                     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, `plex.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.

*Index***A**

- absolute objective difference
 - in integrality constraints of a MIP **256**
 - in MIP performance **280**
- absolute optimality tolerance
 - definition **280**
 - gap **280**
- accessing
 - basis information (C++ API) **57**
 - current parameter value (C API) **122**
 - current parameter value (C++ API) **53**
 - default parameter value (C API) **121**
 - dual values (C++ API) **57**
 - maximum parameter value (C API) **121**
 - maximum parameter value (Java API) **85**
 - minimum parameter value (C API) **121**
 - minimum parameter value (Java API) **85**
 - objective function value (C++ API) **57**
 - reduced costs (C++ API) **57**
 - solution quality (C++ API) **58**
 - solution values (C++ API) **56**
- active model
 - as instance of `IloCplex` (Java API) **76**
 - MIP (Java API) **82**
- active node **390**
- add method
 - `IloModel` C++ class
 - extensible arrays **47**
 - modifying a model **59**
 - `addMinimize` method (Java API) **78**
- advanced basis
 - example **184**
 - in networks **164**
 - parallel threads and **163**
 - primal feasibility and **163**
 - reading from file (LP) **168**
 - saving to file (LP) **168**
 - starting from (callbacks) **426**
 - starting from (goals) **426**
 - starting from (LP) **168**
- advanced start
 - callbacks **426**
 - goals **426**
- advanced starting basis **184**
- `AdvInd` parameter
 - MIP start **269**
 - solution polishing and **266**
- `AggCutLim` **264**
- `AggCutLim` parameter
 - controlling cuts **264**
- `AggInd` parameter
 - MIP preprocessing **267**
- aggregate goal **394**
- aggregator
 - barrier preprocessing **199**
 - MIP and **267**
 - simplex and **166**
- algorithm
 - choosing in LP (C++ API) **51**

- controlling in IloCplex (C++ API) **53**
- pricing **169**
- type (Java API) **83**
- using multiple **342**
- Algorithm.Barrier (Java API) **87**
- application
 - creating with Concert Technology (C++ API) **43**
 - development steps (C++ API) **43**
- arc **208**
- architecture
 - C++ API **42**
 - Callable Library (C API) **108**
 - Java API **70**
- arguments
 - null pointers (C API) **116**
 - optional (C API) **116**
- array
 - constructing (Java API) **90**
 - creating multi-dimensional (C++ API) **65**
 - creating variables in (Java API) **74**
 - extensible (C++ API) **47**
 - using for I/O (C++ API) **65**

B

- BarAlg
 - large objective values and **206**
 - log file and **195**
 - settings **203**
- barrier optimizer
 - algorithm **188**
 - algorithms and infeasibility **206**
 - barrier display parameter **203**
 - centering corrections **203**
 - column nonzeros parameter and density **200**
 - column nonzeros parameter and instability **204**
 - corrections limit **203**
 - growth parameter **205**
 - infeasibility analysis **206**
 - inhibiting dual formulation **166**
 - linear **187 to 206**
 - log file **192**
 - numeric difficulties and **204**
 - numerical emphasis and **202**
 - parallel **451**

- performance tuning **197**
- preprocessing **199**
- primal-dual **164**
- QCP and **232**
- quadratic **217 to 228**
- quadratic constraints and **232**
- row-ordering algorithms **200**
- second-order cone program (SOCP) and **232**
- simplex optimizer and **189**
- solution quality **195**
- solving LP problems **187**
- starting-point heuristics **201**
- unbounded optimal face and **205**
- unbounded problems **205**
- uses **189**
- BarStartAlg parameter
 - barrier starting algorithm **201**
- BarThreads parameter
 - overriding thread limit in parallel **454**
 - parallel processing and **448**
- BAS file format **178**
- basis
 - accessing information (C++ API) **57**
 - advanced, primal feasible **163**
 - advanced, role in network **164**
 - advanced, starting from (example) **184**
 - column generation and **336**
 - condition number **177, 180**
 - crash parameter and **172**
 - crossover algorithms **189**
 - current (C API) **109**
 - differences between LP and network optimizers **215**
 - from basis file **215**
 - infeasibility and **179**
 - maximum row residuals and **182**
 - no factoring in network optimizer **214**
 - objective in Phase I and **179**
 - optimal, condition number and **177**
 - optimal, numeric stability and **176**
 - parallel threads and advanced basis **163**
 - preprocessing versus starting (MIP) **269**
 - previous optimal (C API) **111**
 - refactoring rate **171**
 - removing objects from (C++ API) **60**
 - role in converting LP to network flow **216**

- role in converting network-flow to LP **215**
- role in network optimizer **211**
- role in network optimizer to solve QP **212**
- saving best so far **178**
- sensitivity analysis and (Java API) **87**
- singularities and **177**
- starting from advanced **431**
- unexplored nodes in tree (MIP) **281**
- unstable optimal **180**
- BBInterval parameter
 - controlling branch and cut **257**
- Best Bound **258**
- Best Estimate **258**
- bibliography **37**
 - column generation **336**
- Big M **317**
- BndStrenInd parameter
 - MIP preprocessing **267**
- bound violation (LP) **182**
- Bounded return status (Java API) **79**
- branch & cut algorithm
 - controlling process with presolve **431**
 - heuristic callback in **441**
 - Java API **82**
 - memory problems and **281**
 - MIP performance tuning and **255**
 - parallel processing **453**
 - parameters **257**
 - priority order **270**
 - solves MIP models **251**
 - special ordered sets (SOS) **291**
 - storing tree nodes **260, 282**
 - tree **255**
 - tree and presolvenode
 - solving linear problem at **430**
 - tree subproblems **286**
- branch variable selection callback **443**
- branching direction (Java API) **86**
- branching goal **393**
- BrDir parameter
 - controlling branch and cut **257**
- breakpoint
 - discontinuous piecewise linear and **302**
 - example **300**
 - piecewise linear function and **300**

- BtTol parameter
 - backtrack parameter, purpose **257**
 - controlling branch and cut **257**

C

- call by value (C API) **112**
- Callable Library
 - categories of routines **108**
 - core **108**
 - debugging and **137**
 - description **26**
 - parameters **121**
 - using **107 to 128**
- callback
 - branch variable selection **443**
 - control **417**
 - cut **442**
 - diagnostic **408**
 - graphic user interface and **417**
 - heuristic **441**
 - incumbent **444**
 - node selection **445**
 - resetting to null (C API) **123**
 - resetting to null (C++ API) **54**
 - solve **445**
 - using status variables **416**
- changing
 - bounds setLB (Java API) **93**
 - bounds setUB (Java API) **93**
 - limit on barrier corrections **203**
 - maximization to minimization **214**
 - model setLinearCoef (Java API) **93**
 - pricing algorithm **211**
 - problem type
 - network to LP **215**
 - qp **224**
 - zeroed_qp **224**
 - quadratic coefficients **223**
 - type of variable **341**
 - variable type (C++ API) **60**
- channel example **151**
- character string length requirements (C API) **117**
- check.c CPLEX file **118**
- Cholesky factor **197**

- barrier iteration **188**
- barrier log file and **194**
- clique cuts
 - counting **263**
 - definition **262**
- Cliques parameter
 - controlling cuts **264**
- ClockType parameter
 - parallel processing and **450**
- cloneK.log **456**
- clones **456**
 - log files **456**
 - threads and **456**
- closing
 - application (C API) **112**
 - application (network) **214**
 - environment (C API) **112**
 - environment (network) **214**
 - log files **147**
- CoeRedInd parameter
 - MIP preprocessing **267**
- column
 - dense **204**
 - density **200**
 - index number (C API) **116**
 - name (C API) **116**
 - nonzeros parameter and density **200**
 - nonzeros parameter and instability **204**
 - referencing (C API) **116**
- column generation
 - basis and **336**
 - cutting plane method and **336**
 - reduced cost and (example) **338**
 - reduced cost to determine next variable **336**
- columnwise modeling (C API) **124**
- columnwise modeling (C++ API) **64**
- columnwise modeling (Java API)
 - ILoMModeler and **73**
 - objective and **90**
 - ranges and **90**
- complementarity **188**
 - barrier optimizer and **188**
 - convergence tolerance **205**
 - unbounded models and **205**
- Component Libraries (definition) **26**
- Concert Technology
 - accessing parameter values (C++ API) **53**
 - application development steps (C++ API) **43**
 - creating application (C++ API) **43**
 - description **26**
 - design (C++ API) **42**
 - error handling (C++ API) **61**
 - solving problem with (C++ API) **42**
 - using (C++ API) **41 to 68**
 - writing programs with (C++ API) **41**
- concurrent optimizer **452**
 - licensing issues **448**
 - non-default parameters and **165**
 - option for (C++ API) **52**
 - parallel processing and **453**
 - root relaxation and **453**
- cone (SOCP) **232**
- conflict
 - comparing IIS **355**
 - definition **353**
 - groups in **368**
- conflict refiner **353**
 - C++ API example **365**
 - Interactive Optimizer example **356**
 - possible status **363**
 - proved status **363**
- constraint
 - adding with user-written callback **442**
 - convex quadratic **229**
 - creating ranged (Java API) **72**
 - cuts as **261**
 - indicator **317**
 - lazy **381, 442**
 - logical **311**
 - modeling linear (C++ API) **49**
 - quadratic **229**
 - ranged (Java API) **75**
 - removing from basis (C++ API) **60**
 - representing with ILoRange (C++ API) **46**
 - violation **182**
- constructing arrays of variables (Java API) **90**
- continuous piecewise linear **301**
- continuous relaxation (Java API) **82**
- continuous relaxation (MIP) **255**
- continuous relaxation subproblem **390**

- control callback
 - definition **417**
 - types of **417**
- conventions
 - character strings (C API) **117**
 - naming **144**
 - notation **33**
 - numbering **140, 142**
 - numbering rows, columns **139**
- convergence tolerance
 - barrier algorithm and **165**
 - definition **197**
 - effect of tightness **204**
 - performance and **197**
- convert CPLEX utility **145**
- converting
 - error code to string **213**
 - file formats **145**
 - network-flow model to LP **214**
 - network-flow problem to LP **215, 216**
- convex
 - quadratic constraints and **230**
- convex quadratic constraint **229**
- cover cuts **262**
 - counting **263**
 - defined **262**
- Covers parameter
 - controlling cuts **264**
- CPLEX
 - Component Libraries **26**
 - core (C API) **108**
 - licensing (C++ API) **42**
 - parameters (C++ API) **53**
- cpex.h header file
 - C API **122**
 - extern statements in **121**
 - in an application **139**
 - macros for pointers in **119**
- cpex.log file
 - changing name **192**
 - clone logs **456**
 - default name **147, 170**
- CPX_ALG_AUTO symbolic constant **287**
- CPX_ALG_CONCURRENT symbolic constant **287**
- CPX_ALG_DUAL symbolic constant **287**

- CPX_ALG_HYBNETOPT symbolic constant **287**
- CPX_ALG_PRIMAL symbolic constant **287**
- CPX_ALG_SIFTING symbolic constant **287**
- CPX_INTEGER_INFEASIBLE **444**

CPX_PARAM_A

- CPX_PARAM_ADVIND
 - MIP start **269**
 - presolve and advanced start **435**
 - solution polishing and **267**
- CPX_PARAM_AGGIND
 - MIP preprocessing **267**

CPX_PARAM_B

- CPX_PARAM_BARSTARTALG
 - barrier starting algorithm **201**
- CPX_PARAM_BARTHREADS
 - parallel processing and **448**
- CPX_PARAM_BBINTERVAL
 - controlling branch and cut **257**
- CPX_PARAM_BNDSTRENIND
 - MIP preprocessing **267**
- CPX_PARAM_BRDIR
 - controlling branch and cut **257**
- CPX_PARAM_BTTOL
 - controlling branch and cut **257**

CPX_PARAM_C

- CPX_PARAM_CLIQUES
 - controlling cuts **264**
- CPX_PARAM_CLOCKTYPE
 - example of parameter checking **122**
 - parallel processing and **450**
- CPX_PARAM_COEREDIND
 - MIP preprocessing **267**
- CPX_PARAM_COVERS
 - controlling cuts **264**
- CPX_PARAM_CUTLO
 - conflict refiner and **354**
 - FeasOpt and **372**
- CPX_PARAM_CUTUP
 - conflict refiner and **354**

FeasOpt and **372**

CPX_PARAM_D

CPX_PARAM_DATACHECK
entering problem data and **117**

CPX_PARAM_DEPIND
barrier **199**
LPs and **167**

CPX_PARAM_DISJUNCTIONS
controlling cuts **264**

CPX_PARAM_E

CPX_PARAM_EPAGAP
limiting MIP optimization **254**

CPX_PARAM_EPGAP
limiting MIP optimization **254**

CPX_PARAM_EPOPT **182**

CPX_PARAM_EPRHS **182**

CPX_PARAM_F

CPX_PARAM_FLOWCOVERS
controlling cuts **264**

CPX_PARAM_FLOWPATHS
controlling cuts **264**

CPX_PARAM_FRACCUTS
controlling cuts **264**

CPX_PARAM_G

CPX_PARAM_GUBCOVERS
controlling cuts **264**

CPX_PARAM_I

CPX_PARAM_IMPLBD
controlling cuts **264**

CPX_PARAM_INTSOLLIM
limiting MIP optimization **254**

CPX_PARAM_L

CPX_PARAM_LPMETHOD

choosing LP optimizer **162**

network flow **211**

parallel processing and **451**

CPX_PARAM_M

CPX_PARAM_MEMORYEMPHASIS
barrier **198**

conserving memory **173**
final factor after preprocessing **167**
presolve and **167**

CPX_PARAM_MIPCBREDLP
branch callbacks and **444**
callback arguments and **441**
heuristic callbacks and **441**
incumbent callback and **444**
presolved and original problem **441**
user defined cuts and **443**

CPX_PARAM_MIPTHREADS
parallel processing and **448**

CPX_PARAM_MIRCUTS
controlling cuts **264**

CPX_PARAM_N

CPX_PARAM_NODEFILEIND
effect on storage **283**
node files and **283**

CPX_PARAM_NODELIM
limiting MIP optimization **254**

CPX_PARAM_NOSESEL
controlling branch and cut **257**

CPX_PARAM_NUMERICALEMPHASIS
barrier **202**
LP **175**

CPX_PARAM_P

CPX_PARAM_POLISHTIME
solution polishing **266**

CPX_PARAM_PREIND
MIP preprocessing **267**

CPX_PARAM_PRELINEAR
advanced MIP control and **442**
advanced presolve **434**

- user cut pools **383**
- user defined cuts **383**
- CPX_PARAM_PREPASS
 - MIP preprocessing **267**
- CPX_PARAM_PROBE
 - MIP **261**
- CPX_PARAM_QPMETHOD
 - network flow and quadratic objective **212**

CPX_PARAM_R

- CPX_PARAM_REDUCE
 - advanced presolve **433**
 - infeasible problems and **434**
 - lazy constraints and **383**
 - lazy constraints and advanced MIP control **442**
 - MIP preprocessing **267**
 - optimal basis and **435**
 - presolve and problem modifications **437**
 - problem modifications and **435**
 - unbounded problems and **434**
- CPX_PARAM_RELAXPREIND
 - advanced presolve **430**
 - MIP preprocessing **267**
- CPX_PARAM_REPAIRTRIES
 - MIP starts and **269**
- CPX_PARAM_REPEATPRESOLVE
 - MIP preprocessing **267**
 - purpose **268**

CPX_PARAM_S

- CPX_PARAM_SCAIND **171**
- CPX_PARAM_SCRIND
 - error checking and **118**
 - example lpex6.c **185**
 - example with callbacks **416**
 - managing input and output **150**
 - network flow **213**
 - programming practices and **139**
 - repeated singularities and **177**
- CPX_PARAM_SCRIND parameter
 - data checking and **118**
 - reporting repeated singularities **177**
- CPX_PARAM_STARTALG

- controlling algorithm in initial relaxation (MIP) **260**
- initial subproblem and **286**
- parallel processing and barrier **453**
- CPX_PARAM_STRONGTHREADLIM
 - parallel processing and **448**
- CPX_PARAM_SUBALG
 - controlling algorithm at nodes **260**
 - node relaxations and **287**
- CPX_PARAM_SUBMIPNODELIM
 - solution polishing and **267**

CPX_PARAM_T

- CPX_PARAM_THREADS
 - parallel processing and **448**
- CPX_PARAM_TILIM
 - limiting MIP optimization **254**
 - solution polishing and **266**
- CPX_PARAM_TRELIM
 - effect on storage **283**
 - limiting MIP optimization **254**
 - node files and **283**

CPX_PARAM_V

- CPX_PARAM_VARSEL
 - controlling branch and cut **257**

CPX_PARAM_W

- CPX_PARAM_WORKDIR
 - barrier **199**
 - node file subdirectory **284**
 - node files and **283**
- CPX_PARAM_WORKMEM
 - barrier **198**
 - node files and **283**
- CPX_PREREDUCE_DUALONLY **433**
- CPX_PREREDUCE_NO_PRIMALORDUAL **433**
- CPX_PREREDUCE_PRIMALANDDUAL **433**
- CPX_PREREDUCE_PRIMALONLY **433**
- CPX_SEMICONT **296**
- CPX_SEMIINT **296**

CPXa

CPXaddchannel routine
 data types in Callable Library and **114**
 message handling and **150**

CPXaddcols routine
 maintainable code and **133**
 memory management and **115**
 modifying problems **437**

CPXaddfpdest routine
 example lpex5.c **151**
 file pointers and **119**
 message channels and **150**

CPXaddfunctest routine
 example **151**
 function pointers and (C API) **119**
 message channels and **150**

CPXaddindcontr **318**

CPXaddrows routine
 example **125**
 memory allocation and (C API) **115**
 modularity and **133**

CPXaddusercuts **433**

CPXALG_BARRIER symbolic constant **287**

CPXb

CPXbaropt **435**

CPXbasicpresolve **436**

CPXc

CPXCENVptr **114**

CPXCHANNELptr data type **114**

CPXCHARptr data type **119**

CPXcheckaddcols routine **118**

CPXcheckaddrows routine **118**

CPXcheckchgcoeflist routine **118**

CPXcheckcopyctype routine **118**

CPXcheckcopylp routine **118**

CPXcheckcopylpnames routine **118**

CPXcheckcopyqsep routine **118**

CPXcheckcopyquad routine **118**

CPXcheckcopysos routine **118**

CPXcheckvals routine **118**

CPXchgbds **437**

CPXchgcoeflist routine **133**

CPXchgprobtype routine **288**

CPXchgqpcoef routine **223**
 changing quadratic terms **223**
 example **224**

CPXchgrhs **437**

CPXcloneprob routine
 advanced preprocessing and **436**
 copying node LPs **442**

CPXcloseCPLEX routine
 callbacks and **416**
 example lpex6.c **185**
 example mipex2.c **289**
 example qpex1.c **227**
 example qpex2.c **228**
 managing input and output **151**
 network flow problems **214**
 purpose **112**

CPXCCLPptr **114**

CPXCNETptr **114**

CPXcopybase **435**

CPXcopybase routine **185**

CPXcopyctype routine **296**

CPXcopyctype routine
 checking types of variables **138**
 example mipex1.c **288**
 specifying types of variables **247**

CPXcopylp routine **111, 133**

CPXcopynettolp routine **214**

CPXcopyorder routine **294**

CPXcopyprotected **434, 444**

CPXcopyquad routine **228**

CPXcopysos routine
 example mipex3.c **294**

CPXcopystart **435, 436**

CPXcopystart routine
 advanced presolved solution and **435**
 crushing primal or dual solutions **435**

CPXcreateprob **416**

CPXcreateprob routine **228**
 data types and **114**
 example lpex6.c **185**
 example mipex1.c **288**
 example mipex2.c **289**

example qpex1.c **227, 228**
problem object (C API) **110**
role in application **125**
CPXcutcallbackadd **433, 442**

CPXd

CPXdelchannel routine **150, 151**
CPXdelfpdest routine **119, 150, 151**
CPXdelfuncdest routine **150, 151**
CPXdelindconstr **318**
CPXdisconnectchannel routine **150**
CPXdualopt **435, 445**

CPXe

CPXENVptr data type **114**
CPXERR_NEGATIVE_SURPLUS symbolic constant **126**
CPXERR_PRESLV_INF **434**
CPXERR_PRESLV_UNBD **434**
CPXERR_PRESOLVE_BAD_PARAM **383**
cpxerror message channel **149, 151**

CPXf

CPXfclose routine **119**
CPXFILEptr data type **119**
CPXflushchannel routine **150**
CPXfopen routine **119, 147**
CPXfputs routine **119**
CPXfreepresolve **433**
CPXfreeprob **433**
CPXfreeprob routine **112, 185, 227, 228, 289, 416**

CPXg

CPXgetcallbackglobalb **441, 442**
CPXgetcallbackglobalub **441**
CPXgetcallbackincumbent **441**
CPXgetcallbackinfo routine **118, 414, 415, 416**
CPXgetcallbacklp **441, 442**
CPXgetcallbacknodeintfeas **443, 444**
CPXgetcallbacknodelb **441**
CPXgetcallbacknodelp **441**
CPXgetcallbacknodeub **441**

CPXgetcallbacknodex **442**
CPXgetcallbackorder **443**
CPXgetcallbackpseudocosts **443**
CPXgetchannels routine **114, 149, 151**
CPXgetcolindex routine **117**
CPXgetcolname routine **127, 128**
CPXgetcols routine **126, 127**
CPXgetctype routine **248**
CPXgetdblparam routine **116, 122**
CPXgetdblquality routine **177, 183, 195**
CPXgeterrorstring routine **213, 416**
CPXgetintparam routine **116, 122**
CPXgetintquality routine **195**
CPXgetnumcols routine **115**
CPXgetobjval routine **288**
CPXgetredlp **436**
CPXgetrowindex routine **117**
CPXgetrowname routine **115**
CPXgetslack routine **288**
CPXgetsos routine **248**
CPXgetstat routine **288, 415**
CPXgetstrparam routine **116, 122**
CPXgetx routine **113, 288**

CPXh

CPXhybnetopt **445**
CPXinfodblparam routine **116, 121**
CPXinfointparam routine **116, 121**
CPXinfostrparam routine **116, 121, 122**

CPXI

cpxlog message channel **149**
CPXlpopt **227, 228**
CPXlpopt routine **125, 417**
CPXLPptr data type **114**

CPXm

CPXmemcpy routine **120**
CPXMIP_ABORT_FEAS symbolic constant **415**
CPXMIP_ABORT_INFEAS symbolic constant **415**
CPXmipopt **435**
CPXmipopt routine **288, 289**

CPXmsg routine **110, 119, 150, 151**
CPXmsgstr routine **120**

CPXn

CPXNETaddarcs routine **213**
CPXNETaddnodes routine **213**
CPXNETcheckcopynet routine **118**
CPXNETchgobjsen routine **214**
CPXNETcreateprob routine **114, 213**
CPXNETdelnodes routine **214**
CPXNETfreeprob routine **214**
CPXNETprimopt routine **214, 216**
CPXNETptr data type **114**
CPXNETsolution routine **214**
CPXnewcols routine **125, 133**
CPXnewrows routine **133**

CPXo

CPXopenCPLEX routine
 data types and **114**
 example lpex1.c **151**
 example lpex6.c **185**
 example netex1.c **213**
 example qpex1.c **227**
 example qpex2.c **228**
 initializing environment **109**
 managing input and output **149**
 parameters and **122**
 role in application **125**
CPXordwrite routine **294**

CPXp

CPXpreaddrows **437**
CPXpresolve **435, 436**
CPXprimopt **435**
CPXprimopt routine **121, 288**
CPXPROB_FIXEDMILP symbolic constant **288**
CPXPUBLIC symbolic constant **119**
CPXPUBVARARGS symbolic constant **119**
CPXqpopt routine **227, 228**

CPXr

CPXreadcopyprob routine **111, 128**
cpxresults message channel **149**

CPXs

CPXsavwrite routine **136**
CPXsetbranchcallbackfunc **443**
CPXsetcutcallbackfunc **442**
CPXsetdblparam routine **116, 122**
CPXsetdefaults routine **123**
CPXsetheuristiccallbackfunc **441**
CPXsetintparam routine
 arguments of **122**
 example lpex6.c **185**
 example netex1.c **213**
 parameter types and **116**
 redirecting output to screen **139**
 selecting root algorithm **286**
 setting clock type **450**
CPXsetlogfile routine **147, 192**
 channels and **150**
 collecting messages **118**
 file pointers and **119**
 managing log files **147**
CPXsetlpcallbackfunc routine **119, 415, 417**
CPXsetmipcallbackfunc routine **119, 415**
CPXsetnodecallbackfunc **445**
CPXsetsolvecallbackfunc **445**
CPXsetstrparam routine **116, 122**
CPXsolution routine **126, 288, 415**
CPXstrcpy routine **120**
CPXstrlen routine **120**

CPXv

CPXVOIDptr data type **119**

CPXw

cpxwarning message channel **149**
CPXwriteprob routine **126, 137, 178, 294**

C (continued)

creating
 application with Concert Technology (C++ API) **43**
 array of variables (Java API) **74**
 arrays of variables (Java API) **74**
 Boolean variables (Java API) **74**
 CPLEX environment **213**
 log file **147**
 modeling variables (Java API) **72, 74**
 network flow problem object **213**
 new rows (Java API) **89**
 objective function (Java API) **72**
 problem object (C API) **110**
 ranged constraints (Java API) **72**

crossover
 verifying barrier solutions **202**

CSV file format **146**

cut callback **442**

CutLo parameter **256**
 conflict refiner and **354**
 FeasOpt and **372**

cuts **262, 390**
 adding **263**
 clique **262**
 counting **263**
 cover **262**
 disjunctive **262**
 dual reductions and **433**
 flow cover **262**
 flow path **262**
 Gomory fractional **262**
 GUB cover **263**
 implied bound **263**
 local or global **391**
 MIR **263**
 recorded in MIP node log file **274**
 re-optimizing **263**
 what are **261**

CutsFactor parameter
 controlling cuts **264**

cutting plane method **336**

CutUp parameter **256**
 conflict refiner and **354**
 FeasOpt and **372**

D

data
 entering (C API) **110**

data types
 special (C API) **114**

debugging
 Callable Library and **137**
 diagnostic routines and (C API) **118**
 heap **140**
 Interactive Optimizer and **136**
 return values and **139**

definition **246**

degeneracy
 dual **286**
 stalling and **178**

delete goal stacks **398**

deleting
 model objects (C++ API) **59**
 variables (Java API) **93**

deleting nodes **398**

dense column **200**

dense matrix
 reformulating QP **221**

DepInd parameter
 barrier **199**
 LPs and **167**

Depth First **258**

destroying
 CPLEX environment (C API) **112**
 nodes **214**
 problem object (C API) **112**

devex pricing **171**

diagnosing
 infeasibility (barrier) **206**
 infeasibility (LP) **179**
 infeasibility (preprocessor) **347**
 infeasibility (QP) **225**
 infeasibility as conflict **353**
 performance problems (LP) **173**
 unboundedness **349, 351**

diagnostic callback
 definition **408**
 types of **408**

diagnostic routine

- C API **118**
 - log file and (C API) **118**
 - message channels and (C API) **118**
 - redirecting output from (C API) **118**
- diet model (Java API) **77**
- diff method (Java API) **74**
- dimensions, checking **140**
- discontinuous piecewise linear **302**
 - breakpoints and **302**
 - segments and **302**
- DisjCuts parameter
 - controlling cuts **264**
- disjunctive cuts **262**
- displaying
 - barrier information **192**
 - barrier solution quality **195**
 - basis condition **177**
 - bound infeasibilities **181**
 - column-nonzeros parameter **205**
 - infeasibilities on screen **181**
 - messages **150**
 - MIP information periodically **275**
 - network objective values **210**
 - network solution information **213**
 - network solution on screen **214**
 - optimization progress **417**
 - problem dimensions **140**
 - problem statistics **140**
 - reduced-cost infeasibilities **181**
 - simplex solution quality **197**
 - solution quality **180**
 - solutions on screen **151**
 - variables **142**
- DUA file format **352**
- dual feasibility **188**
- dual reduction **347**
- dual residual **181**
- dual simplex optimizer
 - perturbing objective function **178**
 - selecting **163**
 - stalling **178**
- dual variable
 - solution data (C++ API) **57**
- duality gap **188**

E

- elapsed time for the MIP optimizer **455**
- emphasis
 - memory (barrier) **198**
 - memory (LP) **173**
 - MIP **251**
 - numerical (barrier) **202**
 - numerical (LP) **175**
- empty goal **393, 397**
- end method
 - IloEnv C++ class **44**
- enter Interactive Optimizer command **247**
- entering **247**
 - LPs for barrier optimizer **190**
 - mixed integer programs (MIPs) **247**
 - network arcs **213**
 - network data **213**
 - network data from file **216**
 - network nodes **213**
- enumeration
 - Algorithm (C++ API) **51**
 - BasisStatus (C++ API) **57**
 - BoolParam (C++ API) **53**
 - IntParam (C++ API) **53**
 - NumParam (C++ API) **53**
 - Quality (C++ API) **58**
 - Status (C++ API) **55**
 - String Param (C++ API) **53**
- environment
 - constructing (C++ API) **44**
 - initializing (C API) **109**
 - multithreaded (C API) **110**
 - releasing (C API) **112**
- EpAGap parameter
 - limiting MIP optimization **254**
 - near zero objective value and **280**
- EpGap parameter
 - limiting MIP optimization **254**
 - when to change **280**
- EpOpt parameter **182**
- EpRHS parameter **182**
- eq method (Java API) **75**
- error checking
 - diagnostic routines for (C API) **118**

- MPS file format **144**
- problem dimensions **140**
- error handling
 - in Concert Technology (C++ API) **61**
 - querying exceptions **139**
- Error return status (C++) **56**
- Error return status (Java API) **79**
- example
 - Column Generation **335**
 - columnwise modeling (C API) **124**
 - columnwise modeling (C++ API) **64**
 - conflict refiner (Interactive Optimizer) **356**
 - creating multi-dimensional arrays (C++ API) **65**
 - Cutting Stock **335**
 - FORTRAN **120**
 - message handler **150**
 - MIP node log file **274**
 - MIP optimization **288**
 - MIP problem from file **289**
 - MIP with SOS and priority orders **293**
 - network optimization **209**
 - optimizing QP **226**
 - output channels **151**
 - Piecewise Linear **299**
 - project staffing **356**
 - reading QP from file **227, 228**
 - resource allocation **356**
 - rowwise modeling (C API) **124**
 - rowwise modeling (C++ API) **63**
 - using arrays for I/O (C++ API) **65**
- executing a goal **391**
- expression
 - building (C++ API) **45**
 - editable (Java API) **75**
 - in ranged constraints (Java API) **76**
 - linear (C++ API) **45**
 - logical (C++ API) **45**
 - piecewise linear (C++ API) **45**
 - square method (Java API) **74**
 - sum method (Java API) **74**
 - using modeling variables to construct (Java API) **72**
- external variables (C API) **112**
- extra rim vectors **143**

F

- FailGoal **392**
- feasibility
 - analysis and barrier optimizer **206**
 - check **391**
 - dual **172, 188**
 - network flows and **209**
 - primal **188**
 - progress toward **179, 209**
- feasibility tolerance
 - largest bound violation and **182**
 - network optimizer and **211**
 - reducing **180**
- Feasible return status (C++) **56**
- Feasible return status (Java API) **79**
- FeasOpt **371**
 - definition **371**
 - example **373**
 - invoking **372**
 - preferences **373**
- feasOpt method
 - C++ API **372**
 - Java API **88**
- file format
 - BAS **178**
 - converting **144**
 - converting fixed-format MPS **145**
 - CSV **146**
 - data in C API **111**
 - described **142**
 - DUA and dual transformation **352**
 - example of quadratic program **227**
 - lazy constraints in LP file **385**
 - LP file and indicator constraints **318**
 - LP file and SOS **293**
 - LP for QCP **233**
 - LP special considerations **142**
 - MIP data and **248**
 - MPS and CPLEX extensions **143**
 - MPS and lazy constraints **386**
 - MPS and quadratic objective **219**
 - MPS and SOS declarations **293**
 - MPS and user cuts **386**
 - MPS for QCP **233**

- MST (Java API) **87**
- MST for MIP starts **269**
- NET **214**
- ORD **270**
- programming considerations and **142**
- QCP and **233**
- SAV for QCP **233**
- SOL and barrier optimizer **192**
- SOL and slack values **271**
- user cuts in LP file **385**
- VEC (Java API) **86**
- file reading routines in Callable Library **109**
- file writing routines in Callable Library **109**
- flow cover cuts
 - defined **262**
- flow path cuts
 - defined **262**
- FlowCovers parameter
 - controlling cuts **264**
- FlowPaths parameter
 - controlling cuts **264**
- FORTTRAN **120, 140**
- FracCand **265**
- FracCand parameter
 - controlling cuts **265**
- FracCuts parameter
 - controlling cuts **264**
- FracPass **265**
- FracPass parameter
 - controlling cuts **265**
- fractional cuts
 - defined **262**
- free row **143**
- free variable
 - MPS files and **143**
 - reformulating QPs and **221**

G

- ge method (Java API) **75**
- generalized upper bound (GUB) cover cuts **263**
- getBasisStatus method
 - IloCplex Java class **87**
- getBasisStatuses method
 - IloCplex C++ class **57**

- getBoundSA method
 - IloCplex C++ class **57**
- getBoundSA method (Java API) **87**
- getCplexStatus method
 - IloCplex C++ class **56, 58**
- getDefault method
 - IloCplex C++ class **53**
- getDual method
 - IloCplex C++ class **57**
- getDual method (Java API) **87**
- getDuals method
 - IloCplex C++ class **57**
- getDuals method (Java API) **87**
- getMax method
 - IloCplex C++ class **53**
- getMax method (Java API) **85**
- getMessage method
 - IloException class **139**
- getMin method
 - IloCplex C++ class **53**
- getMin method (Java API) **85**
- getNumVar method
 - IloCplex class (Java API) **89**
- getObjSA method
 - IloCplex C++ class **57**
- getObjSA method (Java API) **87**
- getObjValue method
 - IloCplex C++ class **57**
- getParam method
 - IloCplex C++ class **53**
- getParam method (Java API) **85**
- getQuality method
 - IloCplex C++ class **58**
- getQuality method (Java API) **85**
- getRange method
 - IloCplex class **177, 195**
- getRange method (Java API) **89**
- getRangeSA method (Java API) **87**
- getReducedCost method
 - IloCplex C++ class **57**
- getReducedCost method (Java API) **87**
- getReducedCosts method
 - IloCplex C++ class **57**
- getReducedCosts method (Java API) **87**
- getRHSSA method
 - IloCplex C++ class **57**

- getSlack method
 - IloCplex C++ class **57**
- getSlacks method
 - IloCplex C++ class **57**
- getStatus method
 - IloCplex C++ class **55**
 - IloCplex::Exception class **139**
- getStatuses method
 - IloCplex class **184**
- getValue method
 - IloCplex C++ class **56**
- getValues method
 - IloCplex C++ class **57**
- global thread limit **454**
- global thread parameter **453**
- global variables (C API) **112**
- goal
 - aggregates **394**
 - And-goal **392**
 - branch as CPLEX **393**
 - empty **393**
 - executing **391**
 - Fail-goal **392**
 - global cuts and **399**
 - local cut **393**
 - Or-goal **392**
 - solution **394**
 - solution injection by **401**
- goal stack **397**
- Gomory fractional cuts
 - defined **262**
- graphic user interface (GUI) **417**
- group **368**
 - definition **368**
 - example in conflict **368**
- GUB
 - constraint **263**
- GUBCovers parameter
 - controlling cuts **264**

H

- head **208**
- header file **139**
- heap, debugging **140**

- heuristic
 - callback **441**
 - definition **265**
 - node **265**
 - relaxation induced neighborhood search (RINS) **265**
 - RINSHeur parameter **265**
 - solutions **401**
 - starting point **201**
 - SubMIPNodeLimand RINS **265**
- histogram
 - column counts **193**
 - detecting dense columns **200**

I

- ill-conditioned
 - basis **182**
 - factors in **183**
 - maximum dual residual and **182**
 - problem **180**
- IloAdd template class (C++ API) **66**
- IloAddable class (Java API)
 - active model **76**
 - modeling objects and **72**
- IloAlgorithm::Exception class (C++ API) **61**
- IloAlgorithm::Status enumeration (C++ API) **55**
- IloArray template class (C++ API) **47**
- IloColumn class
 - and example (Java API) **92**
- IloColumnArray class (Java API) **90**
- IloConstraint class (C++ API) **49**
- IloConversion class (C++ API) **45, 49, 60**
- IloConversion class (Java API) **93**
- IloCplex class
 - getBasisStatus method (Java API) **87**
 - getBasisStatuses method (C++ API) **57**
 - getBoundSA method (C++ API) **57**
 - getCplexStatus method (C++ API) **56, 58**
 - getDefault method (C++ API) **53**
 - getDual method (C++ API) **57**
 - getDuals method (C++ API) **57**
 - getMax method (C++ API) **53**
 - getMin method (C++ API) **53**
 - getObjSA method (C++ API) **57**
 - getObjValue method (C++ API) **57**

- getParam method (C++ API) **53**
- getQuality method **177, 195**
- getQuality method (C++ API) **58**
- getReducedCost method (C++ API) **57**
- getReducedCosts method (C++ API) **57**
- getRHSSA method (C++ API) **57**
- getSlack method (C++ API) **57**
- getSlacks method (C++ API) **57**
- getStatus method (C++ API) **55**
- getStatuses method **184**
- getValue method (C++ API) **56**
- getValues method (C++ API) **57**
- IloMModeler and (Java API) **72**
- isDualFeasible method (C++ API) **56**
- isPrimalFeasible method (C++ API) **56**
- modeling objects and (Java API) **72**
- notifying about changes to (C++ API) **59**
- objects in user application (C++ API) **42**
- PrimalPricing (Java API) **84**
- setDefaultts method (C++ API) **54**
- setParam method (C++ API) **54**
- solve method (C++ API) **50, 55, 56, 57, 58, 59, 60, 67**
- writeBasis method **178**
- IloCplex::Algorithm enumeration **51**
- IloCplex::BasisStatus enumeration (C++ API) **57**
- IloCplex::BoolParam enumeration (C++ API) **53**
- IloCplex::Exception class **139**
 - getStatus method **139**
- IloCplex::Exception class (C++ API) **61**
- IloCplex::IntParam enumeration (C++ API) **53**
- IloCplex::Kappa **177**
- IloCplex::NumParam enumeration (C++ API) **53**
- IloCplex::Quality enumeration (C++ API) **58**
- IloCplex::Reduce **347**
- IloCplex::StringParam enumeration (C++ API) **53**
- IloCplexModeler interface
 - modeling objects (Java API) **72**
- IloCPModeler class (Java API) **72**
- IloEnv class **44**
 - end method (C++ API) **44**
- IloException class
 - getMessage method **139**
- IloExpr C++ class **45**
- ILOG License Manager
 - examples **155**
 - invoking **154**
- ILOG License Manager (ILM) **153**
 - CPLEX (C++ API) and **42**
 - types of **154**
- IloLPMatrix class (Java API) **89**
- IloMaximize C++ function **46**
- IloMinimize C++ function **46, 66**
- IloModel C++ class **46**
- IloModel class
 - add method (C++ API) **47, 59**
 - add method (Java API) **93**
 - remove method (C++ API) **47, 59**
 - remove method (Java API) **93**
- IloModeler class
 - basic modeling (Java API) **74**
 - creating modeling objects (Java API) **72**
 - creating variables (Java API) **74**
- IloMModeler class
 - creating variables (Java API) **74**
- IloMModeler class (Java API) **72**
 - delete method **93**
- IloNumArray C++ class **47**
- IloNumExpr class
 - objective and (Java API) **76**
 - ranged constraints and (Java API) **75**
 - variables and (Java API) **74**
- IloNumVar C++ class **44, 49**
- IloNumVar class
 - modeling objects and (Java API) **72**
- IloNumVarArray C++ class **45**
- IloNumVarclass
 - extension of IloNumExpr (Java API) **74**
- IloObjective C++ class **49**
- IloObjective class
 - addable objects (Java API) **76**
 - as modeling object (C++ API) **49**
 - declaring (C++ API) **45**
 - modeling by column (Java API) **90**
 - setExpr method in QP term **223**
- IloObjectiveSense class
 - example (Java API) **76**
 - maximizing (Java API) **76**
 - minimizing (Java API) **76**
 - objective function and (Java API) **76**
- illoqpex1.cpp example

- example
 - `iloqpex1.cpp` **226**
- `IloRange` class
 - adding constraints (C++ API) **46**
 - linear constraints and (C++ API) **49**
 - modeling by column (Java API) **90**
 - modeling objects and (Java API) **76**
- `IloSemiContVar` class **49**
- `IloSolver` as factory (Java API) **70**
- `IloSOS1` C++ class **49**
- `IloSOS2` C++ class **49**
- `ImplBd` parameter
 - controlling cuts **264**
- implied bound cuts
 - defined **263**
- include file **139**
- incumbent
 - node **256**
 - solution **256**
- incumbent callback **444**
- index number (C API) **116**
- indicator constraint **311**
 - definition **317**
 - restrictions **319**
- indicator variable **319**
- indicators: see indicator constraint
- Individual optimizer parallel processing **454**
- infeasibility
 - barrier optimizer and **206**
 - conflicts and **353**
 - diagnosing in network flows **216**
 - displaying on screen **181**
 - dual **203, 206**
 - interpreting results **181**
 - maximum bound **181, 182**
 - maximum reduced-cost **181, 182**
 - network flow **209**
 - network optimizer and **216**
 - norms **194**
 - primal **195, 203, 206**
 - ratio in barrier log file **195**
 - repairing **371**
 - reports **179**
 - scaling and **180**
 - unboundedness and (LP) **181**

- unscaled **180**
- Infeasible return status (C++ API) **56**
- Infeasible return status (Java API) **79**
- infeasible solution
 - accessing information (Java API) **88**
 - analyzing (C++ API) **57**
- InfeasibleOrUnbounded
 - return status (C++ API) **56**
 - return status (Java API) **79**
- initializing
 - CPLEX environment **213**
 - problem object **213**
 - problem object (C API) **110**
- input operator (C++ API) **47**
- instantiating
 - CPLEX environment **213**
 - problem object **213**
 - problem object (C API) **110**
- integrality constraints **390**
- integrality tolerance
 - MIP **280**
 - parameter **280**
- Interactive Optimizer
 - debugging and **136**
 - description **26**
 - experimenting with optimizers **134**
 - improving application performance **136**
 - testing code in **132**
- `IntSolLim` parameter
 - limiting MIP optimization **254**
- `isDualFeasible` method
 - `IloCplex` C++ class **56**
- isolated point **303**
- `isPrimalFeasible` method
 - `IloCplex` C++ class **56**

J

- Java serialization **72**

K

- knapsack constraint
 - cover cuts and **262**
 - GUB cover cuts and **263**

knapsack problem with reduced cost in objective **337**

L

lazy constraint **442**

definition **381**

Interactive Optimizer and **384**

LP file format and **384, 386**

MPS file format and **386**

pool **381 to 387**

SAV file format and **384, 386**

le method

in expressions (Java API) **75**

license

CPLEX (C++ API) **42**

runtime **153**

limiting

network iterations **211**

strong branching candidate list **279**

strong branching iterations **279**

linear expression (C++ API) **45**

linear objective function (C++ API) **49**

linear relaxation

as initial subproblem in MIP **286**

MIP and coefficients at nodes **268**

MIP and preprocessing **267**

MIP and progress reports **273**

local cut goal **393**

log file

barrier optimizer **192**

Cholesky factor in **194**

clones and **456**

closing **147**

contents **170, 195**

creating **147**

default **147**

description **147**

diagnostic routines and (C API) **118**

iteration **174**

naming **147**

network **211**

parallel MIP optimizer and **456**

parameter **147**

records infeasibilities **181**

records singularities **177**

relocating **147**

renaming **147**

logical constraint **311, 312**

example in early tardy scheduling **329**

logical expression (C++ API) **45**

LP

barrier optimizer **187**

choosing algorithm (C++ API) **51**

network optimizer **207**

problem formulation **26, 188**

solving **161 to 206**

LP file format

lazy constraints **384**

lazy constraints in **385**

QCP and **233**

QPs and **222**

row, column order **142**

SOS declarations in **293**

special considerations **142**

user cuts **384**

user cuts in **385**

M

managing

log file **147**

memory (LP) **173**

memory (MIP) **281**

Markowitz tolerance **178, 179**

default **179**

increasing to stay feasible **179**

maximum value **178**

numeric stability and **178**

pivots and **178**

slow progress in Phase I and **179**

maximal cliques

recorded in MIP node log file **275**

maximization

concave QPs **218**

lower cutoff parameter **280**

maximize method

objective functions and (Java API) **76**

maximum bound infeasibility **182**

maximum reduced-cost infeasibility **182**

maximum row residual **182**

- memory **282**
- memory emphasis
 - barrier **198**
 - continuous (LP) **173**
- memory leaks (C++ API) **44**
- memory management
 - MIPs and **281**
 - performance in LP **173**
 - refactoring frequency and **174**
- MemoryEmphasis parameter
 - barrier **198**
 - conserving memory and **173**
 - final factor after preprocessing **167**
 - presolve and **167**
- message channel
 - diagnostic routines and (C API) **118**
- message handler (example) **150**
- minimal covers
 - recorded in MIP node log file **275**
- minimization
 - convex QPs **218**
 - upper cutoff parameter **280**
- minimize method
 - objective functions and (Java API) **76**
- MIP **245 to 289**
 - active model (Java API) **82**
 - branch & cut (Java API) **82**
 - changing variable type **250**
 - memory problems and **281**
 - optimizer **245**
 - problem formulation **246**
 - progress reports **273**
 - relaxation algorithm **286**
 - subproblem algorithm **286**
 - subproblems **285**
 - supplying first integer solution **269**
 - terminating optimization **253**
- MIP file format **219**
- MIP gap tolerance **253**
 - absolute **253**
 - relative **253**
- MIPEmphasis **251**
- MIPThreads parameter
 - parallel processing and **448**
- MipThreads parameter
 - branch and cut tree **454**
- MIR cuts **263**
- MIRCuts parameter
 - controlling cuts **264**
- Mixed Integer Linear Program (MILP)
 - definition **246**
 - definition (Java API) **82**
- Mixed Integer Programming (MIP)
 - definition **246**
- Mixed Integer Quadratic Program (MIQP)
 - definition **246**
 - definition (Java API) **82**
- Mixed Integer Quadratically Constrained Program (MIQCP) **246**
- model
 - active (Java API) **76**
 - adding columns to **340**
 - adding objects (C++ API) **59**
 - adding submodels (C++ API) **46**
 - changing variable type **341**
 - consistency and tolerance **421**
 - deleting objects (C++ API) **59**
 - extracting (C++ API) **50**
 - IloMPSModeler and (Java API) **73**
 - modifying (Java API) **92**
 - notifying changes to IloCplex object (C++ API) **59**
 - portfolio optimization **221**
 - reformulating dense QP **221**
 - reformulating large QP **221**
 - removing objects (C++ API) **59**
 - serializing **145**
 - solving (C++ API) **42, 50**
 - solving with IloCplex (C++ API) **67**
 - XML representation of **145**
- modeling
 - columnwise (C API) **124**
 - columnwise (C++ API) **64**
 - objects (C++ API) **42**
 - rowwise (C API) **124**
 - rowwise (C++ API) **63**
- modeling by column (Java API)
 - IloMPSModeler and **73**
 - objective and **90**
 - ranges and **90**
- modeling variable

- creating (Java API) **72**
- `IloNumVar` (Java API) **74**
- modifying
 - constraints in QCP **240**
 - model (Java API) **92**
- MPS file format
 - conversion utility **145**
 - CPLEX extensions **143**
 - lazy constraints in **386**
 - quadratically constrained program (QCP) in **233**
 - saving modifications **144**
 - saving QP **222**
 - SOS declarations in **293**
 - user cuts in **386**
- MST file format **87, 269**
- multithreaded application
 - needs multiple environments (C API) **110**

N

- namespace conflicts (C API) **112**
- naming
 - arcs in network flow **213**
 - conventions **144**
 - log file **147**
 - node file **284**
 - nodes in network flow **213**
- negative method
 - expressions and (Java API) **74**
- negative semi-definite objective **219**
- nested parallel processing **454**
- NET file format **214**
- `NetItLim` **211**
- network
 - converting to LP model **214**
 - embedded **211**
 - infeasibility in **209**
 - modeling variables **208**
 - problem formulation **208, 209**
- network extractor **212**
- network object **208**
- network optimizer **164, 207 to 213**
 - preprocessing and **213**
 - problem formulation **209**
 - turn off preprocessing **213**

- node **390**
 - demand **209**
 - from **208**
 - head **208**
 - sink **209**
 - source **209**
 - supply **209**
 - tail **208**
 - to **208**
 - transshipment **209**
 - viable **441**
- node file **282**
 - cpx name convention **284**
 - parameters and **283**
 - using with MIP **260**
 - when to use **260, 282**
- node heuristic **265**
- node log **273**
- node problem **390**
- node selection callback **445**
- node selection strategy
 - best estimate **285**
 - depth-first search **285**
- `NodeAlg`
 - controlling algorithm at subproblems (MIP) **260**
- `NodeAlg` parameter
 - node relaxations and **287**
- `NodeFileInd` parameter
 - effect on storage **283**
 - node files and **283**
- `NodeLim` parameter
 - limiting MIP optimization **254**
- `NodeSel` parameter
 - controlling branch and cut **257**
- nondeterminism **449**
- nonlinear expression
 - definition **314**
- nonseparable **218**
- notation in this manual **33**
- notifying
 - changes to `IloCplex` object (C++ API) **59**
- null goal **393**
 - definition **393**
 - when to use **393**
- numbering conventions

- C 140**
 - Fortran **140**
 - row, column order **142**
- numeric difficulties
 - barrier growth parameter **205**
 - barrier optimizer and **204**
 - basis condition number and **177**
 - complementarity **205**
 - convergence tolerance **205**
 - definition (LP) **174**
 - dense columns removed **204**
 - infeasibility and **179**
 - sensitivity **177**
 - unbounded optimal face **205**
- numeric variable (C++ API) **49**
- numerical emphasis
 - barrier optimizer and **202**
 - continuous (LP) **175**
- NumericalEmphasis parameter
 - barrier **202**
 - LP **175**

O

- ObjDif tolerance parameter **256**
- objective coefficients
 - crash parameter and **172**
 - modified in log file **210**
 - network flows and **210**
 - priority and **271**
- objective difference
 - absolute **256, 280**
 - relative **256, 280**
- objective function
 - accessing value of (C++ API) **57**
 - changing sense **214**
 - constructor (Java API) **76**
 - creating (Java API) **72**
 - free row as **143**
 - in log file **210**
 - in MPS file format **144**
 - maximization **144**
 - maximize (C++ API) **46**
 - minimize (C++ API) **46**
 - modeling (Java API) **76**

- network flows and **209**
- optimality tolerance and **182**
- preprocessing and **347**
- primal reductions and **347**
- representing with IloObjective (C++ API) **45**
- sign reversal in **144**
- objective value
 - accessing slack in (C++ API) **57**
 - in log file **210**
 - network flows and **209**
 - object range parameter **205**
 - unbounded **205**
- operator << (C++ API) **47**
- operator >> (C++ API) **47**
- Optimal return status (C++ API) **56**
- Optimal return status (Java API) **79**
- optimality
 - basis condition number and **177**
 - cutoff parameters **280**
 - infeasibility ration **195**
 - normalized error and **197**
 - singularities and **178**
 - tolerance **180, 182**
 - relative **280**
- optimality tolerance
 - absolute **280**
 - changing relative or absolute **280**
 - gap **280**
 - maximum reduced-cost infeasibility and **182**
 - Network and **211**
 - reducing **180**
 - relative **280**
 - relative, default **280**
 - setting **182**
 - when to change **279**
- optimization
 - interrupting **417**
 - stopping **253, 417**
- optimization problem
 - defining with modeling objects (C++ API) **42**
 - representing with IloModel (C++ API) **46**
- optimization routines in Callable Library **108**
- optimizer
 - barrier (linear) **187 to 206**
 - barrier (quadratic) **217 to 228**

- choosing (Java API) **81, 83**
- concurrent **452**
- differences between Barrier, simplex **189**
- dual simplex **163**
- MIP **245**
- network **164, 207 to 213**
- parallel **447 to 456**
- primal simplex **164**
- primal-dual barrier **164**
- optimizing
 - cuts **263**
- ORD file format **270**
- OrGoal **397**
- output
 - channel parameter **148**
 - debugging and **139**
 - redirecting **149**
- output operator (C++ API) **47**

P

- parallel
 - license **448**
 - optimizers **447 to 456**
 - threads **448**
- Parallel Barrier Optimizer **451**
- Parallel MIP Optimizer **452**
 - memory considerations **454**
 - output log file **454**
- parallel processing
 - branch & cut **453**
 - individual optimizer **454**
 - nested **454**
 - root relaxation **453**
 - selected starting algorithm **453**
- parameter
 - accessing
 - current value (C API) **122**
 - current value (C++ API) **53**
 - current value (Java API) **85**
 - default value (C API) **121**
 - default value (C++ API) **53**
 - default value (Java API) **85**
 - maximum value (C API) **121**
 - maximum value (C++ API) **53**

- maximum value (Java API) **85**
 - minimum value (C API) **121**
 - minimum value (C++ API) **53**
 - minimum value (Java API) **85**
- algorithmic **197**
- barrier corrections **203**
- Callable Library and **121**
- classes of (Java API) **84**
- controlling branch & cut strategy **257**
- gradient **169**
- log file **147**
- NetFind network extractor **212**
- object range **205**
- optimality cutoff **280**
- output channel **148**
- preprocessing dependency **199**
- routines in Callable Library **109**
- screen indicator **213**
- setting
 - all defaults (C API) **123**
 - all defaults (C++ API) **54**
 - all defaults (Java API) **85**
 - branching direction (Java API) **86**
 - C API **122**
 - C++ API **54**
 - example algorithm (Java API) **83**
 - example steepest edge pricing (Java API) **85**
 - example turn off presolve (Java API) **84**
 - Java API **84**
 - priority in MIP (Java API) **86**
 - RootAlg (Java API) **83**
- symbolic constants as (C API) **122**
- tree memory **282**
- types of
 - C API **122**
 - C++ API **54**
 - Java API string **84**
 - Java API StringParam **84**
- performance
 - Barrier
 - centering corrections and **203**
 - characteristics **189**
 - dense columns and **200**
 - memory management and **198**
 - numeric difficulties and **204**

- preprocessing and **199**
- row order and **200**
- tuning **197**
- convergence tolerance and **197**
- LP
 - advanced basis and **168**
 - automatic selection of optimizer **163**
 - increasing available memory **174**
 - network as model **164**
 - numeric difficulties and **164**
 - parameters for **169**
 - preprocessing and **166**
 - preprocessing and memory **174**
 - refactoring and **174**
 - troubleshooting **173**
 - tuning **165**
- MIP
 - default optimizer and **251**
 - feasibility emphasis **251**
 - node files and **282**
 - probing and **261**
 - RINS and **266**
 - subproblems and **286**
 - swap space and **260**
 - troubleshooting **277**
 - tuning **254**
- negative impact of Reduce parameter **348**
- Network optimizer
 - general observations **208**
 - tuning **211**
- QP
 - reformulating for **221**
 - tuning **224**
- SOS
 - branching strategies and **292**
- perturbation constant (LP) **179**
- perturbing
 - objective function **178**
 - variable bounds **178**
- piecewise linear **299**
 - continuous **301**
 - definition **300**
 - discontinuous **302**
 - example **300**
 - example in early tardy scheduling **329**

- expression (C++ API) **45**
- IlomPModeler and (Java API) **73**
- isolated point ignored **303**
 - steps **302**
- polishing a solution **266**
- PolishTime parameter
 - solution polishing **266**
- pool
 - of cuts **382**
 - of lazy constraints **382**
 - of user cuts **382**
- populating problem object **213**
- populating problem object (C API) **110**
- portability (C API) **119**
- portfolio optimization model **221**
- positive semi-definite
 - objective **219**
 - quadratic constraint **232**
 - second-order cone program (SOCP) and **232**
- possible status in conflict refiner **363**
- preference
 - example **368**
 - FeasOpt **373**
- PreInd parameter
 - MIP preprocessing **267**
- PreLinear parameter
 - user cut pools **383**
 - user defined cuts **383**
- PrePass parameter
 - MIP preprocessing **267**
- preprocessing
 - advanced basis and (LP) **169**
 - barrier and **199**
 - barrier optimizer **199**
 - definition of **166**
 - dense columns removed **204**
 - dependency parameter **199**
 - dual reductions in **347**
 - lazy constraints and **383**
 - MIPs **267**
 - network optimizer and **213**
 - primal reductions in **347**
 - second-order cone program (SOCP) and **232**
 - simplex and **166**
 - starting-point heuristics and **201**

- turning off **168**
- presolve **430**
 - barrier preprocessing **199**
 - dependency checking in **166**
 - final factorization after uncrush in **167**
 - gathering information about **435**
 - interface **434**
 - lazy constraints and **383**
 - limited **436**
 - process for MIP **430**
 - protecting variables during **434**
 - restricting dual reductions **433**
 - restricting primal reductions **433**
 - simplex and **166**
 - simplex preprocessing **166**
 - turning off (Java API) **84**
- presolved problem
 - adding constraints to **431**
 - and branch & cut process **440**
 - building **430**
 - freeing **433**
 - freeing memory **436**
 - retaining **433**
- pricing algorithms **211**
- primal feasibility **188**
- primal reduction **347**
- primal simplex optimizer **164**
 - perturbing variable bounds **178**
 - stalling **178**
- primal variables **172**
- primal-degenerate problem **163**
- priority **271**
 - binary variables and **270**
 - integer variables and **270**
 - order **270**
 - parameter to control **271**
 - reading from file **270**
 - semi-continuous variables and **270**
 - semi-integer variables and **270**
 - special ordered set (SOS) and **270**
- priority order (Java API) **86**
- Probe parameter
 - MIP **261**
- probing parameter **260**
- problem
 - analyzing infeasible (C++ API) **57**
 - solving with Concert Technology (C++ API) **42**
- problem description
 - example: Rates **296**
 - example: semi-continuous variables **296**
 - example:Column Generation **337**
 - example:Cutting Stock **337**
- problem formulation
 - barrier **188**
 - dual **188, 190**
 - ill-conditioned **180**
 - infeasibility reports **179**
 - linear **26**
 - network **209**
 - network-flow **208**
 - primal **188, 190**
 - removing dense columns **200**
 - switching from network to LP **214, 216**
- problem modification routines in Callable Library **108**
- problem object
 - creating (C API) **110**
 - destroying (C API) **112**
 - freeing (C API) **112**
 - initializing (C API) **110**
 - instantiating (C API) **110**
 - network **208**
 - populating **213**
 - populating (C API) **110**
- problem query routines in Callable Library **108**
- problem representation
 - example: Rates **297**
 - example: semi-continuous variables **297**
 - example:Column Generation **338**
 - example:Cutting Stock **338**
- problem solution
 - example: Rates **298**
 - example: semi-continuous variables **298**
 - example:Column Generation **342**
 - example:Cutting Stock **342**
- problem type
 - changing from network to LP **215, 216**
 - changing to qp **224**
 - changing to zeroed_qp **224**
 - quadratic programming and **222**
- prod method in expressions (Java API) **74**

proved status in conflict refiner **363**
pruned node **390**
PSD
 positive semi-definite in objective function **219**
 quadratic constraints and **232**
 second-order cone program (SOCP) as exception to **232**

Q

QCP

barrier optimizer and **232**
convexity and **230**
determining problem type **233**
examples **241**
file types and **233**
modifying constraints in **240**
PSD and **232**

QP

example **226, 227, 228**
portfolio optimization **221**
problem formulation **218**
reformulating large, dense models **221**
solution example **227, 228**
solving **217 to 228**

QP relaxation **225**

quadratic

constraints **229**
convex constraints **229**

quadratic coefficient

changing **223**

quadratic objective function (C++ API) **49**

quadratically constrained programming (QCP) **229 to 241**

queried **85**

query routine (C API) **127**

R

ranged constraint

creating (Java API) **72**
definition (Java API) **75**
name of (Java API) **75**

ranged row **143**

reading

MIP problem data **289**
MIP problem data from file **248**

network data from file **216**

QP problem data from file **227, 228**

start values from MST file **269**

redirecting

diagnostic routines (C API) **118**
log file output **149**
output **139**
screen output **149**

Reduce parameter

lazy constraints and **383**
MIP preprocessing **267**

reduced cost

accessing (C++ API) **57**
accessing (Java API) **87**
choosing variables in column generation **338**
column generation and **336**
pricing (LP) **171**

reduction

dual **347**

reduction, primal **347**

refactoring frequency

dual simplex algorithm and **165**
primal simplex algorithm and **165**

reference counting **398**

reference row values **293**

refineConflict

Java API **88**

reflection scaling **212**

relative objective difference **256, 280**

relative optimality tolerance

default (MIP) **280**
definition **280**

relaxation

algorithm applied to **286**
of MIP problem **255**
QP **225**

solving MIPs (Java API) **82**

relaxation induced neighborhood search (RINS) **265**

RelaxPreInd parameter

advanced presolve **430**
MIP preprocessing **267**

RelObjDif tolerance parameter **256**

relocating log file **147**

remove method

IloModel C++ class **47, 59**

- renaming
 - log file **147**
- repairing
 - infeasibility **371**
- RepairTries parameter
 - MIP starts and **269**
- RepeatPresolve parameter
 - MIP preprocessing **267**
 - purpose **268**
- residual
 - dual **181**
 - maximum dual **182**
 - maximum row **182**
 - row **181**
- return status
 - Bounded (Java API) **79**
 - Error (C++) **56**
 - Error (Java API) **79**
 - Feasible (C++) **56**
 - Feasible (Java API) **79**
 - Infeasible (C++) **56**
 - Infeasible (Java API) **79**
 - InfeasibleOrUnbounded (C++ API) **56**
 - InfeasibleOrUnbounded (Java API) **79**
 - Optimal (C++ API) **56**
 - Optimal (Java API) **79**
 - Unbounded (C++ API) **56**
 - Unbounded (Java API) **79**
 - Unknown (C++ API) **56**
 - Unknown (Java API) **79**
- return value
 - C API **115**
 - debugging with **139**
 - routines to access parameters (C API) **122**
- righthand side (RHS)
 - file formats for **143**
- rim vectors **143**
- RINSHeur parameter **265**
- root relaxation
 - parallel processing **453**
- RootAlg parameter
 - controlling initial relaxation algorithm **260**
 - initial subproblem and **286**
 - network flow **211**
 - network flow and quadratic objective **212**
 - parallel processing and **451**
 - parallel processing and barrier **453**
- row
 - index number (C API) **116**
 - name (C API) **116**
 - referencing (C API) **116**
 - residual **181**
- row-ordering algorithms **200**
 - approximate minimum degree (AMD) **200**
 - approximate minimum fill (AMF) **200**
 - automatic **200**
 - nested dissection (ND) **200**
- rowwise modeling
 - C API **124**
 - C++ API **63**

S

- SAV file format **222**
 - lazy constraints **386**
 - QCP and **233**
 - user cuts **386**
- saving
 - best factorable basis **178**
- ScaInd parameter **171**
- scaled problem statistics **181**
- scaling
 - alternative methods of **171**
 - definition **171**
 - feasibility and **180**
 - in network extraction **212**
 - infeasibility and **180**
 - maximum row residual and **182**
 - numeric difficulties and QP **226**
 - objective function in QP **226**
 - singularities and **178**
 - tolerance and **178**
- search limit **406**
- search tree **390**
- second order cone programming (SOCP) **229**
- second-order cone program (SOCP)
 - formulation **232**
- semi-continuous variable
 - C++ API **49**
 - example **297**

- Java API **73**
- priority and **270**
- semi-definite
 - negative and objective **219**
 - positive and constraints **232**
 - positive and objective **219**
- semi-integer variable **296**
 - priority and **270**
- sensitivity analysis (C++ API) **57**
- sensitivity analysis (Java API) **87**
- separable **218**
- serialization **72**
- serializing **145**
- setDefault method
 - IloCplex C++ class **54**
- setExpr method
 - IloObjective class **223**
- setOut **147**
- setParam method
 - IloCplex C++ class **54**
- setting
 - algorithm in LP (C++ API) **51**
 - all default parameters (C API) **123**
 - all default parameters (C++ API) **54**
 - callbacks to null (C API) **123**
 - callbacks to null (C++ API) **54**
 - parameters (C API) **122**
 - parameters in C++ API **54**
- sifting **164**
- simplex
 - column generation and **336**
 - dual **163**
 - feasibility tolerance in MIP **281**
 - optimizer **189**
 - pricing phase and **336**
 - primal **164**
- simplex method
 - column generation and **336**
 - pricing phase and **336**
- singularity **177**
- slack
 - accessing bound violations in (C++ API) **58**
 - accessing in constraints in active model (Java API) **80**
 - accessing slack variables in constraints (C++ API) **57**
 - accessing slack variables in objective (C++ API) **57**
 - as indicator of ill-conditioning **183**
 - as reduced cost in infeasibility analysis **183**
 - example CPXgetslack **288**
 - in primal formulation (Barrier) **188**
 - in summary statistics **181**
 - infeasibilities as bound violations and **182**
 - infeasibility in dual associated with reduced costs **182**
 - initial norms and **170**
 - maximum bound violation and (Java API) **88**
 - meaning in infeasible primal or dual LP **182**
 - pivoted in when constraint is removed (C++ API) **60**
 - primal bound error in solution quality (Barrier) **196**
 - reducing computation of steepest edge pricing **171**
 - role in inequality constraints (Barrier) **194**
 - role in infeasibility analysis **183**
 - row complementarity in solution quality (Barrier) **196**
 - steepest edge and (dual) **170**
 - steepest edge and (primal) **170**
 - using primal variables instead **172**
 - variable needed in basis (Network) **215**
 - variables and primal variables (dual) **172**
- SOCP second-order cone program **232**
- SOL file format **192, 271**
- solution
 - accessing quality information (C++ API) **58**
 - accessing values of (C++ API) **56**
 - basic infeasible primal **179**
 - basis **189**
 - complementary **188**
 - differences between barrier, simplex **189**
 - example QP **227, 228**
 - feasible in MIPs **269**
 - incumbent **256**
 - infeasible basis **206**
 - midface **189**
 - nonbasis **189**
 - quality **195, 202**
 - serializing **145**
 - supplying first integer in MIPs **269**
 - using advanced presolved **435**
 - verifying **202**
 - XML representation of **145**
- solution goal **394**
- solution polishing **266**
- solve callback **445**

- solve method
 - IloCplex C++ class **50, 55, 56, 57, 58, 59, 60, 67**
- solving
 - diet problem (Java API) **80**
 - model (C++ API) **50**
 - single LP (Java API) **82**
 - subsequent LPs or QPs in a MIP (Java API) **83**
- sparse matrix
 - IloLPMatrix and (Java API) **89**
- special ordered set (SOS)
 - role in model (Java API) **73**
 - type 1 (C++ API) **49**
 - type 2 (C++ API) **49**
 - using **291**
 - weights in **293**
- speed increase **453**
- stalling **178**
- starting algorithm
 - callbacks and **426**
 - goals and **426**
 - parallel processing **453**
- static variables (C API) **112**
- status variables, using **416**
- steepest-edge pricing **171, 282**
- step in piecewise linear function **302**
- stopping criterion
 - callbacks and **417**
- strong branching **279**
- StrongThreadLim **453**
- StrongThreadLim parameter
 - parallel processing and **448**
- SubMIPNodeLim parameter
 - RINS and **265**
 - solution polishing and **267**
- summary statistics **181**
- suppressing output to the screen **152**
- surplus argument (C API) **126**
- symbolic constants (C API) **115, 122**

T

- tail **208**
- terminating
 - because of singularities **178**
 - MIP optimization **253**

- network optimizer iterations **211**
- The **254**
- threads **448**
 - clones **456**
 - parallel optimizers **448**
 - performance and **449**
- Threads global parameter **453**
- Threads parameter
 - parallel processing and **448**
- thread-safe (C API) **112**
- TiLim parameter
 - limiting MIP optimization **254**
 - solution polishing and **266**
- time limit
 - concurrent optimizer and **165**
 - effects all algorithms invoked by concurrent optimizer **165**
 - possible reason for Unknown return status (C++ API) **56**
 - possible reason for Unknown return status (Java API) **79**
 - TiLim parameter (MIP) **252**
- tolerance
 - absolute objective difference and **256**
 - absolute optimality **280**
 - advice about **421**
 - complementarity convergence, default of **205**
 - complementary solution and **188**
 - consistency in model and **421**
 - convergence and barrier algorithm **165**
 - convergence and numeric inconsistencies **204**
 - convergence and performance **197**
 - cut callbacks and **421**
 - cut callbacks and (example) **421**
 - cuts in goals and **400**
 - default numeric (example LP) **176**
 - feasibility (Network) **211**
 - feasibility and largest bound violation **182**
 - feasibility, reducing **180**
 - integrality
 - example (Java API) **93**
 - Markowitz **178**
 - Markowitz and numeric difficulty **179**
 - Markowitz, increasing to stay feasible **179**
 - optimality **182**
 - optimality (Network) **211**
 - optimality, reducing **180**

- relative objective difference and **256**
- relative optimality **280**
- relative optimality default **280**
- role of (C++ API) **58**
- role of (Java API) **88**
- simplex feasibility in cut callback **421**
- simplex optimality (example C++ API) **54**
- singularities and scaling **178**
- termination and **253**
- violated constraints in goals and **400**
- warning about absolute and relative objective difference **256**
- when reducing does not help **180**
- TreLim parameter
 - effect on storage **283**
 - limiting MIP optimization **254**
 - node files and **283**
- type
 - changing for variable (Java API) **73**
 - conversion (Java API) **93**

U

- unbounded optimal face
 - barrier optimizer **190**
 - detecting **205**
- Unbounded return status (C++ API) **56**
- Unbounded return status (Java API) **79**
- unboundedness **349**
 - dual infeasibility and **182**
 - infeasibility and **182**
 - infeasibility and (LP) **181**
 - optimal objective and **181**
 - unbounded ray and **350**
- Unknown return status (C++ API) **56**
- Unknown return status (Java API) **79**
- unscaled problem statistics **181**
- user cut
 - definition **381**
 - Interactive Optimizer and **384**
 - LP file format and **384, 386**
 - pool **381 to 387**
 - SAV file format and **384, 386**
- user cuts
 - MPS file format and **386**

- utility routines in Callable Library **108**

V

- variable
 - accessing dual (C++ API) **57**
 - changing type (C++ API) **45, 60**
 - changing type of **341**
 - constructing arrays of (Java API) **90**
 - creating modeling (Java API) **72**
 - deleting (Java API) **93**
 - external (C API) **112**
 - global (C API) **112**
 - in expressions (C++ API) **45**
 - modeling (Java API) **74**
 - not addable (Java API) **76**
 - numeric (C++ API) **49**
 - order **142, 143**
 - removing from basis (C++ API) **60**
 - representing with ILoNumVar (C++ API) **44**
 - semi-continuous (C++ API) **49**
 - semi-continuous (example) **297**
 - semi-continuous (Java API) **73**
 - semi-integer **296**
 - static (C API) **112**
 - type **247**
- variable selection strategy
 - strong branching **279, 285**
- variable type change (Java API) **73**
- VarSel **453**
- VarSel parameter
 - controlling branch and cut **257**
- VEC file format **86**
- vectors, rim **143**
- violation
 - bound **182**
 - constraint **182**

W

- WorkDir parameter
 - barrier **199**
 - node file subdirectory **284**
 - node files and **283**
- working directory

- barrier **199**
- working memory
 - barrier **198**
- WorkMem **282**
- WorkMem parameter
 - barrier **198**
 - node files and **283**
- writeBasis method
 - IloCplex class **178**

X

XML

- Concert Technology and **145**
- serializing model, solution **145**