



Changing the rules of business™

ILOG CPLEX 11.2

ILOG CPLEX

NOTICES

Product copyright notice

ILOG CPLEX Copyright © 1997 – 2008, by ILOG SA, 9 Rue de Verdun, 94253 Gentilly Cedex, France, and ILOG, Inc., 1195 W. Fremont Avenue, Sunnyvale, California 94087-3832, USA. All rights reserved.

Restrictions of general use

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 SA 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 USA and/or other countries.

All other brand, product, and company 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.

Acknowledgement of use: dtoa routine of the gdtoa package

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

<http://www.netlib.org/fp/>.

The author of this software is David M. Gay.

All Rights Reserved.

Copyright (C) 1998, 1999 by Lucent Technologies

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies and that both that the copyright notice and this permission notice

and warranty disclaimer appear in supporting documentation, and that the name of Lucent or any of its entities not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

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

(end of license terms of `dtoa` routine of the `gdtoa` package)

Table of contents

ILOG CPLEX Release Notes.....	14
ILOG CPLEX 11.2 Release Notes.....	15
Welcome to ILOG CPLEX 11.2.....	17
Port changes: deprecated and removed ports.....	18
Conversion notes for all users.....	19
New solution polishing interface.....	21
Impact of multiple MIP starts.....	31
MIP starts independent of the solution pool.....	35
Change in XML schema.....	41
Lower and upper cutoff tolerances.....	42
User cuts may be purged.....	43
Conversion notes for users of MS Windows.....	44
Conversion notes for users of Concert Technology.....	45
Conversion notes for users of C++ API.....	46
Conversion notes for users of Java API.....	47
Conversion notes for users of .NET API.....	49
Conversion notes for users of the Callable Library.....	51
New features for all users.....	55
New parameters.....	57
New error codes.....	58
Solving more SOCP, QCP, and MIQCP.....	59
Multiple MIP starts.....	60
Write level for SOL, MST file formats.....	64

Refining a conflict in a MIP start.....	65
Accessing MIP relative gap.....	66
New ways to measure computational time.....	67
ILOG CPLEX 11.1 Release Notes.....	69
Welcome to ILOG CPLEX 11.1.....	71
Announcements.....	72
Port changes.....	73
Conversion notes for all users.....	77
Conversion overview.....	78
Solution polishing and MIP starts.....	79
Locale and the writing of readable files.....	80
Conversion notes for Concert Technology users.....	81
Conversion notes for Callable Library users.....	82
ILOG CPLEX 11.0.....	83
ILOG CPLEX 11.0 Release Notes.....	85
Announcements: port changes.....	86
Conversion notes for all users.....	87
Overview.....	89
Changes in existing parameters.....	90
Dynamic search.....	91
Informational callbacks.....	92
Node limits.....	93
File format removed: SOS.....	94
Error codes removed.....	95
Error codes added.....	96
Conversion notes for Concert Technology users.....	97
Overview.....	98
Conversion notes for users of the C++ API.....	99
Conversion notes for users of the Java API.....	100
Conversion notes for users of the .NET API.....	101
Conversion notes for Callable Library users.....	102
New features for all users.....	105
Overview.....	107
Solution pool.....	108
Tuning tool.....	109
Deterministic parallel MIP.....	110
New algorithm for MIQCPs.....	111
Feasibility pump.....	112
New status code for FeasOpt.....	113
New status codes for solution pool.....	114
Solution information available for MIPs and FeasOpt.....	115
Interruption and termination.....	116
New parameters.....	117

- New features in Concert Technology.....119
 - Overview.....120
- New features in the Callable Library.....121
- New features in the Interactive Optimizer.....122
- Documentation for IDEs.....123
- Getting Started with ILOG CPLEX.....124**
- ILOG CPLEX Getting Started.....125**
- Introducing ILOG CPLEX.....127
 - What is ILOG CPLEX?.....129
 - Using the parallel optimizers.....135
 - Data entry options.....136
 - What ILOG CPLEX is not.....137
 - What you need to know.....138
 - What's in this manual.....139
 - Notation in this manual.....140
 - Related documentation.....141
- Setting up ILOG CPLEX.....143
 - Overview.....144
 - Installing ILOG CPLEX.....145
 - Setting up licensing.....148
 - Using the Component Libraries.....149
- Tutorials.....151
 - Solving an LP with ILOG CPLEX.....153
 - Interactive Optimizer tutorial.....165
 - Concert Technology tutorial for C++ users.....237
 - Concert Technology tutorial for Java users.....279
 - Concert Technology tutorial for .NET users.....301
 - Callable Library tutorial.....315
- ILOG CPLEX User's Manual.....349**
- ILOG CPLEX User's Manual.....351**
- Meet ILOG CPLEX.....353
 - What is ILOG CPLEX?.....355
 - What does ILOG CPLEX do?.....356
 - What you need to know.....358
 - Examples online.....359
 - Notation in this manual.....361
 - Related documentation.....362
 - Announcements and updates.....365
 - Further reading.....366
- Languages and APIs.....367
 - ILOG Concert Technology for C++ users.....369
 - ILOG Concert Technology for Java users.....423

ILOG Concert Technology for .NET users.....	473
ILOG CPLEX Callable Library.....	503
Programming considerations.....	553
Developing CPLEX applications.....	555
Managing input and output.....	583
Timing interface.....	607
Licensing an application.....	611
Tuning tool.....	625
Continuous optimization.....	643
Solving LPs: simplex optimizers.....	645
Solving LPs: barrier optimizer.....	691
Solving network-flow problems.....	733
Solving problems with a quadratic objective (QP).....	751
Solving problems with quadratic constraints (QCP).....	775
Discrete optimization.....	799
Solving mixed integer programming problems (MIP).....	801
Solution pool: generating and keeping multiple solutions.....	911
Using special ordered sets (SOS).....	969
Using semi-continuous variables: a rates example.....	977
Using piecewise linear functions in optimization: a transport example.....	987
Logical constraints in optimization.....	1013
Indicator constraints in optimization.....	1025
Using logical constraints: Food Manufacture 2.....	1033
Early tardy scheduling.....	1043
Using column generation: a cutting stock example.....	1055
Infeasibility and unboundedness.....	1075
Preprocessing and feasibility.....	1077
Managing unboundedness.....	1081
Diagnosing infeasibility by refining conflicts.....	1085
Repairing infeasibilities with FeasOpt.....	1121
Advanced programming techniques.....	1131
User-cut and lazy-constraint pools.....	1133
Using goals.....	1149
Using optimization callbacks.....	1183
Goals and callbacks: a comparison.....	1231
Advanced presolve routines.....	1235
Advanced MIP control interface.....	1251
Parallel optimizers.....	1269
ILOG CPLEX Parameters.....	1290
ILOG CPLEX Parameters Reference Manual.....	1291
Accessing parameters.....	1292
Parameter names.....	1293
Correspondence of parameters.....	1294

Saving parameter settings to a file.....	1295
Topical list of parameters.....	1297
Simplex.....	1299
Barrier.....	1300
MIP.....	1301
MIP general.....	1302
MIP strategies.....	1303
MIP cuts.....	1304
MIP tolerances.....	1305
MIP limits.....	1306
Solution polishing.....	1307
Solution pool.....	1308
Network.....	1309
Parallel optimization.....	1310
Sifting.....	1311
Preprocessing: aggregator, presolver.....	1312
Tolerances.....	1313
Limits.....	1314
Display and output.....	1315
List of CPLEX parameters.....	1317
advanced start switch.....	1329
constraint aggregation limit for cut generation.....	1331
preprocessing aggregator fill.....	1332
preprocessing aggregator application limit.....	1333
barrier algorithm.....	1334
barrier column nonzeros.....	1335
barrier crossover algorithm.....	1336
barrier display information.....	1337
convergence tolerance for LP and QP problems.....	1338
barrier growth limit.....	1339
barrier iteration limit.....	1340
barrier maximum correction limit.....	1341
barrier objective range.....	1342
barrier ordering algorithm.....	1343
convergence tolerance for QC problems.....	1344
barrier starting point algorithm.....	1345
MIP strategy best bound interval.....	1346
bound strengthening switch.....	1347
MIP branching direction.....	1348
backtracking tolerance.....	1349
MIP cliques switch.....	1351
clock type for computation time.....	1352
coefficient reduction setting.....	1353

variable (column) read limit.....	1354
conflict information display.....	1355
MIP covers switch.....	1356
simplex crash ordering.....	1357
lower cutoff.....	1359
number of cutting plane passes.....	1360
row multiplier factor for cuts.....	1361
upper cutoff.....	1362
data consistency checking switch.....	1363
dependency switch.....	1364
MIP disjunctive cuts switch.....	1365
MIP dive strategy.....	1366
dual simplex pricing algorithm.....	1367
type of cut limit.....	1368
absolute MIP gap tolerance.....	1370
relative MIP gap tolerance.....	1371
integrality tolerance.....	1372
epsilon used in linearization.....	1373
Markowitz tolerance.....	1375
optimality tolerance.....	1376
perturbation constant.....	1377
relaxation for FeasOpt.....	1378
feasibility tolerance.....	1379
mode of FeasOpt.....	1380
MIP flow cover cuts switch.....	1382
MIP flow path cut switch.....	1383
feasibility pump switch.....	1384
candidate limit for generating Gomory fractional cuts.....	1386
MIP Gomory fractional cuts switch.....	1387
pass limit for generating Gomory fractional cuts.....	1388
MIP GUB cuts switch.....	1389
MIP heuristic frequency.....	1390
MIP implied bound cuts switch.....	1391
MIP integer solution limit.....	1392
simplex maximum iteration limit.....	1393
local branching heuristic.....	1394
memory reduction switch.....	1395
MIP callback switch between original model and reduced, presolved model.....	1396
MIP node log display information.....	1398
MIP emphasis switch.....	1400
MIP node log interval.....	1402
MIP priority order switch.....	1403
MIP priority order generation.....	1404

MIP dynamic search switch.....	1405
MIQCP strategy switch.....	1407
MIP MIR (mixed integer rounding) cut switch.....	1409
precision of numerical output in MPS and REW file formats.....	1410
network logging display switch.....	1411
network optimality tolerance.....	1412
network primal feasibility tolerance.....	1413
simplex network extraction level.....	1414
network simplex iteration limit.....	1415
network simplex pricing algorithm.....	1416
MIP subproblem algorithm.....	1417
node storage file switch.....	1419
MIP node limit.....	1420
MIP node selection strategy.....	1421
numerical precision emphasis.....	1422
nonzero element read limit.....	1423
absolute objective difference cutoff.....	1424
lower objective value limit.....	1425
upper objective value limit.....	1426
parallel mode switch.....	1427
simplex perturbation switch.....	1430
simplex perturbation limit.....	1431
absolute MIP gap before starting to polish a feasible solution.....	1432
relative MIP gap before starting to polish a feasible solution.....	1433
MIP integer solutions to find before starting to polish a feasible solution.....	1434
nodes to process before starting to polish a feasible solution.....	1435
time before starting to polish a feasible solution.....	1436
time spent polishing a solution (deprecated).....	1437
limit on number of solutions generated for solution pool.....	1438
primal simplex pricing algorithm.....	1440
presolve dual setting.....	1441
presolve switch.....	1442
linear reduction switch.....	1443
limit on the number of presolve passes made.....	1444
node presolve switch.....	1445
simplex pricing candidate list size.....	1446
MIP probing level.....	1447
time spent probing.....	1448
indefinite MIQP switch.....	1449
QP Q-matrix nonzero read limit.....	1450
primal and dual reduction type.....	1451
simplex refactoring frequency.....	1452
relaxed LP presolve switch.....	1453

relative objective difference cutoff.....	1454
frequency to try to repair infeasible MIP start.....	1455
MIP repeat presolve switch.....	1456
RINS heuristic frequency.....	1457
algorithm for continuous problems.....	1458
algorithm for continuous quadratic optimization.....	1460
MIP starting algorithm.....	1461
constraint (row) read limit.....	1463
scale parameter.....	1464
messages to screen switch.....	1465
sifting subproblem algorithm.....	1466
sifting information display.....	1467
upper limit on sifting iterations.....	1468
simplex iteration information display.....	1469
simplex singularity repair limit.....	1470
absolute gap for solution pool.....	1471
limit on number of solutions kept in solution pool.....	1472
relative gap for solution pool.....	1474
solution pool intensity.....	1475
solution pool replacement strategy.....	1477
MIP strong branching candidate list limit.....	1478
MIP strong branching iterations limit.....	1479
limit on nodes explored when a subMIP is being solved.....	1480
symmetry breaking.....	1481
global default thread count.....	1482
optimizer time limit.....	1485
tree memory limit.....	1486
tuning information display.....	1487
tuning measure.....	1488
tuning repeater.....	1489
tuning time limit.....	1490
MIP variable selection strategy.....	1491
directory for working files.....	1493
memory available for working storage.....	1494
write level for MST, SOL files.....	1495
MIP zero-half cuts switch.....	1497

ILOG CPLEX File Formats.....1498

ILOG CPLEX File Formats Reference Manual.....1499

Brief descriptions of file formats.....	1501
Reading and entering file formats in the Interactive Optimizer.....	1505
Saving problems in the Interactive Optimizer.....	1506
LP file format: matrix models.....	1507
MPS file format: industry standard.....	1515

Overview of MPS.....	1516
Records in MPS format.....	1517
Example of MPS file format.....	1523
Special records in MPS files: ILOG CPLEX extensions.....	1525
Overview of MPS extension.....	1527
Objective sense and name in MPS files.....	1528
Integer variables in MPS files.....	1529
Special ordered sets (SOS) in MPS files.....	1531
Quadratic objective information in MPS files.....	1533
Quadratically constrained programs (QCP) in MPS files.....	1535
Indicator constraints in MPS files.....	1536
User defined cuts in MPS files.....	1538
Lazy constraints in MPS files.....	1539
NET file format: network flow models.....	1540
PRM file format: parameter settings.....	1545
BAS file format: advanced basis.....	1546
MST file format: MIP starts.....	1548
ORD file format: priorities and branching orders.....	1550
SOL file format: solution files.....	1551
FLT file format: filter files for the solution pool.....	1553
Overview of FLT.....	1554
Reading and writing filter files.....	1555
Syntax of a filter file.....	1556
Diversity filters.....	1557
Range filters.....	1558
CSV file format: comma separated values.....	1559
XML file format: serialized models and solutions.....	1560
ILOG CPLEX Interactive Optimizer.....	1561
Interactive Optimizer Commands.....	1563
Overview of commands.....	1564
Table of the commands of the Interactive Optimizer.....	1565
Managing parameters in the Interactive Optimizer.....	1579
Saving a parameter specification file.....	1580
Index.....	1583

ILOG CPLEX 11.2 Release Notes

Thank you for installing ILOG CPLEX 11.2. Before using it, please review these notes highlighting improvements and changes in this version.

ILOG CPLEX 11.1 Release Notes

Thank you for installing ILOG CPLEX 11.1. These release notes highlight improvements in ILOG CPLEX 11.1. Please review these notes before using ILOG CPLEX 11.1.

ILOG CPLEX 11.0

Dedicated to Lloyd Clarke (1964–2007)

ILOG CPLEX 11.2 Release Notes

Thank you for installing ILOG CPLEX 11.2. Before using it, please review these notes highlighting improvements and changes in this version.

In this section

Welcome to ILOG CPLEX 11.2

Describes general outlines of ILOG CPLEX 11.2.

Port changes: deprecated and removed ports

Documents available and deprecated ports.

Conversion notes for all users

Describes changes important to all users, identifies new behavior of existing features, and recommends ways to preserve previous behavior, if necessary. When a parameter, method, or routine is deprecated, it is a good idea to migrate to the recommended replacement as soon as possible. The deprecated parameter, method, or routine will be removed in future versions of the product. If migration to the recommended replacement poses a problem, please consult ILOG CPLEX technical support for advice.

Conversion notes for users of MS Windows

Introduces a change in the clock type parameter

Conversion notes for users of Concert Technology

Describes changes of interest to those who use Concert Technology. A new version of Concert Technology (Concert Technology 2.7) accompanies this release of ILOG CPLEX 11.2. This new version requires you to recompile and link your Concert Technology applications.

Conversion notes for users of the Callable Library

Describes changes of interest to those who use the Callable Library (C API).

New features for all users

Announces new features in ILOG CPLEX 11.2.

Welcome to ILOG CPLEX 11.2

ILOG CPLEX 11.2 introduces new features that are designed to make your applications more capable than ever:

- ◆ New control over **solution polishing** allowing the user to specify the conditions for starting and for terminating the polishing of an integer feasible solution
- ◆ Management of **multiple MIP starts**, either separately from or in conjunction with the solution pool.
- ◆ Extension of the **conflict refiner** to handle MIP starts (a help in debugging models).
- ◆ Automatic transformation of additional types of **quadratic** constraints into second order cone programs (**SOCP**).
- ◆ A new **timing interface** to compute the elapsed time of an operation.

Port changes: deprecated and removed ports

For a complete list of machine types and library formats (including version numbers of compilers and JDKs) see the file `yourCPLEXhome/mptable.html` in the distribution of the product. That file contains more detailed descriptions of available ports than do these highlights.

Port name	Platform	Status	Recommended replacement
ultrasparc32_9_8	Sun Studio 8, 32-bit	deprecated and removed ; no longer available	ultrasparc32_9_9 for Sun Studio 9 and higher
ultrasparc64_9_8	Sun Studio 8, 64-bit	deprecated and removed ; no longer available	ultrasparc64_9_9 for Sun Studio 9 and higher
power32_aix5.2_7.0	IBM Power AIX 5.2 IBM XL C/C++ 7.0 32-bit	deprecated : no longer available in the next release	A new port, compatible with IBM Power AIX 5.3 or higher, will be available in the next release.
power64_aix5.2_7.0	IBM Power AIX 5.2 IBM XL C/C++ 7.0 64-bit	deprecated : no longer available in the next release	A new port, compatible with IBM Power AIX 5.3 or higher, will be available in the next release.

Sun Studio 8

The ports named `ultrasparc32_9_8` and `ultrasparc64_9_8`, compatible with Sun Studio 8 and higher, have been **removed** and are not supported in ILOG CPLEX 11.2. The **recommended replacements** are the new ports `ultrasparc32_9_9` and `ultrasparc64_9_9`, compatible with Sun Studio 9 and higher. The Sun Studio 9 ports are available in ILOG CPLEX 11.2.

IBM Power AIX 5

The ports named `power32_aix5.2_7.0` and `power64_aix5.2_7.0`, previously announced as deprecated, are still **deprecated**, and still available in this release. The **recommended replacement** is migration to AIX 5.3 (or a higher version) followed by migration to a new CPLEX port supporting AIX 5.3 (or higher), which will be provided at the time the deprecated port is removed.

Conversion notes for all users

Describes changes important to all users, identifies new behavior of existing features, and recommends ways to preserve previous behavior, if necessary. When a parameter, method, or routine is deprecated, it is a good idea to migrate to the recommended replacement as soon as possible. The deprecated parameter, method, or routine will be removed in future versions of the product. If migration to the recommended replacement poses a problem, please consult ILOG CPLEX technical support for advice.

In this section

New solution polishing interface

Solution polishing, an optional feature for mixed integer programming (MIP) problems, exploits heuristic methods to improve the best integer feasible solution found so far. It works as a final phase of optimization. ILOG CPLEX 11.2 offers a new interface for finer control over this polishing step. This topic introduces the new interface for solution polishing, outlines changes implied by it, and recommends a migration path with examples.

Impact of multiple MIP starts

Outlines changes in behavior due to support for multiple MIP starts.

MIP starts independent of the solution pool

Outlines changes due to multiple MIP starts being independent of the solution pool.

Change in XML schema

Describes a change in the XML schema with respect to solutions.

Lower and upper cutoff tolerances

Announces better documentation of behavior of cutoff tolerances.

User cuts may be purged

Announces purgeable cuts

New solution polishing interface

Solution polishing, an optional feature for mixed integer programming (MIP) problems, exploits heuristic methods to improve the best integer feasible solution found so far. It works as a final phase of optimization. ILOG CPLEX 11.2 offers a new interface for finer control over this polishing step. This topic introduces the new interface for solution polishing, outlines changes implied by it, and recommends a migration path with examples.

In this section

Solution polishing parameters

Documents improvements in solution polishing and recommends migration to the improved API.

Example: control the time spent polishing

Shows how to manage time spent polishing a feasible solution.

Example: switch to polishing after first feasible solution

Recalls previous interface, then shows new interface, followed by a way to reproduce previous behavior.

Example: control solution polishing with gap

Shows how to use a MIP gap as a criterion to control solution polishing.

Solution polishing parameters

ILOG CPLEX 11.2 offers a new interface for finer control over solution polishing. The new interface applies certain existing parameters to solution polishing, adds new parameters to allow better control of solution polishing, and deprecates a parameter no longer needed to enforce a time limit specific to solution polishing.

Existing parameters to control solution polishing

In previous versions of ILOG CPLEX, the only stopping criterion for solution polishing was set by the parameter `PolishTime`, `CPX_PARAMPOLISHTIME` to limit *time spent polishing a solution* (*deprecated*). General stopping criteria, such as the optimizer time limit, absolute MIP gap, relative MIP gap, MIP node limit, or MIP integer solution limit, did not apply to solution polishing in previous releases of the product.

Now, however, ILOG CPLEX 11.2 allows the user to control more finely **when solution polishing terminates**. In other words, the tolerances and limits listed in this table now apply to solution polishing.

General parameters now apply to solution polishing

Reference Manual	Concert Technology	Callable Library	Interactive Optimizer
<i>absolute MIP gap tolerance</i>	EpAGap	CPX_PARAM_EPAGAP	mip tolerances absmipgap
<i>relative MIP gap tolerance</i>	EpGap	CPX_PARAM_EPGAP	mip tolerances mipgap
<i>MIP integer solution limit</i>	IntSolLim	CPX_PARAM_INTSOLLIM	mip limits solutions
<i>MIP node limit</i>	NodeLim	CPX_PARAM_NODELIM	mip limits nodes
<i>optimizer time limit</i>	TiLim	CPX_PARAM_TILIM	timelimit

New parameters to control solution polishing

In addition to those existing parameters that now control the termination of solution polishing, there are also new parameters specific to the **starting conditions for solution polishing**.

With these new parameters, a user can tell CPLEX when to switch from branch & cut to solution polishing. CPLEX is able to switch after it has found a feasible solution and put into place the MIP structures needed for solution polishing. When these two conditions are met (feasible solution and structures in place), CPLEX stops branch & cut and switches to solution polishing whenever the **first** of these starting conditions is met:

- ◆ when CPLEX achieves a specified absolute MIP gap;
- ◆ when CPLEX achieves a specified relative MIP gap;
- ◆ when CPLEX finds a specified number of integer solutions;

- ◆ when CPLEX processes a specified number of nodes;
- ◆ when CPLEX reaches a specified time limit on time spent in optimization.

New parameters to specify starting conditions for solution polishing in Concert Technology

Reference Manual	Concert Technology	CPLEX starts polishing
<i>absolute MIP gap before starting to polish a feasible solution</i>	PolishAfterEpAGap	After achieving this absolute MIP gap.
<i>relative MIP gap before starting to polish a feasible solution</i>	PolishAfterEpGap	After achieving this relative MIP gap.
<i>MIP integer solutions to find before starting to polish a feasible solution</i>	PolishAfterIntSol	After finding this number of integer solutions.
<i>nodes to process before starting to polish a feasible solution</i>	PolishAfterNode	After processing this number of nodes.
<i>time before starting to polish a feasible solution</i>	PolishAfterTime	After optimization spends this amount of time.

New parameters to specify starting conditions for solution polishing in the Callable Library

Reference Manual	Callable Library	CPLEX starts polishing
<i>absolute MIP gap before starting to polish a feasible solution</i>	CPX_PARAM_POLISHAFTEREPAGAP	After achieving this absolute MIP gap.
<i>relative MIP gap before starting to polish a feasible solution</i>	CPX_PARAM_POLISHAFTEREPGAP	After achieving this relative MIP gap.
<i>MIP integer solutions to find before starting to polish a feasible solution</i>	CPX_PARAM_POLISHAFTERINTSOL	After finding this number of integer solutions.
<i>nodes to process before starting to polish a feasible solution</i>	CPX_PARAM_POLISHAFTERNODE	After processing this number of nodes.
<i>time before starting to polish a feasible solution</i>	CPX_PARAM_POLISHAFTERTIME	After optimization spends this amount of time.

New parameters to specify starting conditions for solution polishing in the Interactive Optimizer

Reference Manual	Interactive Optimizer	CPLEX starts polishing
<i>absolute MIP gap before starting to polish a feasible solution</i>	mip polishafter absmipgap	After achieving this absolute MIP gap.
<i>relative MIP gap before starting to polish a feasible solution</i>	mip polishafter mipgap	After achieving this relative MIP gap.

Reference Manual	Interactive Optimizer	CPLEX starts polishing
<i>MIP integer solutions to find before starting to polish a feasible solution</i>	<code>mip polishafter solutions</code>	After finding this number of integer solutions.
<i>nodes to process before starting to polish a feasible solution</i>	<code>mip polishafter nodes</code>	After processing this number of nodes.
<i>time before starting to polish a feasible solution</i>	<code>mip polishafter time</code>	After optimization spends this amount of time.

Deprecated parameter: Time spent polishing a solution

The parameter `PolishTime`, `CPX_PARAM_POLISHTIME` to limit *time spent polishing a solution* (*deprecated*) has been **deprecated** in this version and will be **removed** from a future version of ILOG CPLEX. This deprecated parameter is incompatible with the new parameters offering better control over solution polishing. In particular, attempts to set this deprecated parameter together with any of the new solution polishing parameters raise an error:

1807CPXERR_PARAM_INCOMPATIBLE.

Example: control the time spent polishing

As an example of how to manage time spent polishing a feasible solution, suppose the user wants to solve a problem by spending 100 seconds in branch & cut and an additional 200 seconds in polishing.

In previous versions of CPLEX, a user applied the following commands:

- ◆ The user set the general *optimizer time limit* to 100.0 seconds. That parameter (`TiLim`, `CPX_PARAM_TILIM`) controlled only the time spent in branch & cut.
- ◆ The user set the parameter to limit *time spent polishing a solution (deprecated)* to 200.0 seconds. That parameter (`PolishTime`, `CPX_PARAM_POLISHTIME`) controlled only the time spent in polishing.
- ◆ The user called the optimizer.

However, that procedure entails a difficulty if no feasible solution is found within the first 100 seconds of branch & cut. In such a case, solution polishing will not be executed, since it needs a feasible solution to start. That situation does not arise with the new API for solution polishing.

In contrast, with the new API for solution polishing available in ILOG CPLEX 11.2, a user applies the following commands:

1. Set the general *optimizer time limit* to 300.0 seconds. In ILOG CPLEX 11.2, this parameter now controls the total time spent in branch & cut plus solution polishing.
 - ◆ In Concert Technology, set the parameter `TiLim`.
 - ◆ In the Callable Library, set the parameter `CPX_PARAM_TILIM`.
 - ◆ In the Interactive Optimizer, use the command `set timelimit 300.0`.
2. Establish a starting condition for solution polishing by setting to 100.0 seconds the *time before starting to polish a feasible solution*. This parameter controls the time spent in branch & cut before CPLEX switches to polishing.
 - ◆ In Concert Technology, set the parameter `PolishAfterTime`.
 - ◆ In the Callable Library, set the parameter `CPX_PARAM_POLISHAFTERTIME`.
 - ◆ In the Interactive Optimizer, use the command `set mip polishafter time 100.0`.
3. Call the optimizer.

If CPLEX finds a solution in the first 100 seconds of branch & cut, this improved procedure produces the same results as the deprecated procedure of previous versions. However, if CPLEX does not find a solution in the first 100 seconds, then this procedure continues branch

& cut until it finds a solution, and afterwards switches to polishing. This new procedure guarantees that CPLEX spends at most 300 seconds on the model and that CPLEX applies polishing if a solution is found during that time.

Example: switch to polishing after first feasible solution

Previous interface

In previous versions of CPLEX, in order to find a first solution with branch & cut and improve it with solution polishing, a user had to follow these steps:

- ◆ The user set the MIP integer solution limit parameter to 1 (one).
 - In Concert Technology, a user set the parameter `IntSolLim`.
 - In the Callable Library, a user set the parameter `CPX_PARAM_INTSOLLIM`.
 - In the Interactive Optimizer, a user invoked the command `set mip limits solutions 1`.
- ◆ The user set the (deprecated) polishing time limit parameter to a positive value for the number of seconds, such as 100.0 seconds.
 - In Concert Technology, a user set the parameter `PolishTime`.
 - In the Callable Library, a user set the parameter `CPX_PARAM_POLISHTIME`.
 - In the Interactive Optimizer, a user invoked the command `set mip limit polishtime 100.0`.
- ◆ The user called the optimizer.

New interface

With the new solution polishing interface of ILOG CPLEX 11.2, a user follows these steps in order to find a first feasible solution with branch & cut and to invoke solution polishing to improve that first feasible solution:

1. Set to the value 1 (one) the number of *MIP integer solutions to find before starting to polish a feasible solution*.
 - ◆ In Concert Technology, set the parameter `PolishAfterIntSol`.
 - ◆ In the Callable Library, set the parameter `CPX_PARAM_POLISHAFTERINTSOL`.
 - ◆ In the Interactive Optimizer, use the command `set mip polishafter solutions 1`.
2. Set the *optimizer time limit* to a positive value (for example, 200 seconds) to specify the total time spent in branch & cut plus polishing.
 - ◆ In Concert Technology, set the parameter `TiLim`.
 - ◆ In the Callable Library, set the parameter `CPX_PARAM_TILIM`.
 - ◆ In the Interactive Optimizer, use the command `set timelimit 200.0`.

3. Call the optimizer.

That sequence of steps produces slightly different results compared to the procedure in previous versions of CPLEX. Indeed, the user now controls the **total** time spent in branch & cut and in polishing, instead of controlling only the time spent in polishing.

Reproducing previous behavior

If a user wants to reproduce precisely the results of the old API, the user must call the optimizer twice with different parameters each time, in the new API, like this:

1. Set the general *MIP integer solution limit* to 1 (one).
 - ◆ In Concert Technology, set the parameter `IntSolLim`.
 - ◆ In the Callable Library, set the parameter `CPX_PARAM_INTSOLLIM`.
 - ◆ In the Interactive Optimizer, use the command `set mip limits solutions 1`.
2. Call the optimizer. This call finds the first feasible solution with branch & cut.
3. Reset the general limit on MIP integer solutions to its default value (2 100 000 000).
4. Set the amount of *time before starting to polish a feasible solution* to 0.0 (zero) seconds, so that CPLEX switches immediately to polishing.
 - ◆ In Concert Technology, set the parameter `PolishAfterTime`.
 - ◆ In the Callable Library, set the parameter `CPX_PARAM_POLISHAFTERTIME`.
 - ◆ In the Interactive Optimizer, use the command `set mip polishafter time 0.0`.
5. Set the general *optimizer time limit* to 100.0 seconds to control how much time CPLEX spends polishing.
 - ◆ In Concert Technology, set the parameter `TiLim`.
 - ◆ In the Callable Library, set the parameter `CPX_PARAM_TILIM`.
 - ◆ In the Interactive Optimizer, use the command `set timelimit 100.0`.
6. Call the optimizer again.

Example: control solution polishing with gap

The new API for solution polishing available in ILOG CPLEX 11.2 allows a user to specify when solution polishing starts and ends with respect to a MIP gap. (This degree of control was not possible in the previous API for solution polishing.)

For example, the following procedure applies branch & cut until it reaches a 10% gap; then it starts solution polishing until it narrows the gap to 2%.

1. Set to 10% the tolerance of the *relative MIP gap before starting to polish a feasible solution*. This parameter controls when CPLEX switches from branch & cut to solution polishing.
 - ◆ In Concert Technology, set the parameter `PolishAfterEpGap`.
 - ◆ In the Callable Library, set the parameter `CPX_PARAM_POLISHAFTEREPGAP`.
 - ◆ In the Interactive Optimizer, use the command `set mip polishafter mipgap 0.1`.
2. Set to 2% the *relative MIP gap tolerance*. This parameter controls when branch & cut optimization stops.
 - ◆ In Concert Technology, set the parameter `EpGap`.
 - ◆ In the Callable Library, set the parameter `CPX_PARAM_EPGAP`.
 - ◆ In the Interactive Optimizer, use the command `set mip tolerances mipgap 0.02`.
3. Set the *optimizer time limit* to a positive value (for example, 200 seconds) to specify the total time spent in branch & cut plus polishing. For difficult problems, this step is a precaution to guarantee that CPLEX terminates even if the targeted gap cannot be achieved.
 - ◆ `TiLim` in Concert Technology
 - ◆ `CPX_PARAM_TILIM` in the Callable Library
 - ◆ `timelimit 200.0` in the Interactive Optimizer
4. Call the optimizer.

Impact of multiple MIP starts

Outlines changes in behavior due to support for multiple MIP starts.

In this section

Querying a MIP start or multiple MIP starts

Lists deprecated methods, routines and recommends new methods, routines.

Continuous variables now in MIP starts

Announces feature to support continuous variables in MIP starts.

Querying a MIP start or multiple MIP starts

ILOG CPLEX 11.2 supports **multiple MIP starts** independent of the solution pool. This note highlights conversion issues resulting from this new feature. For more information about the new feature itself, see also *Multiple MIP starts* in these release notes.

As a result of changes to support multiple MIP starts and consequent changes in the solution pool, the query methods and routines for MIP starts have also changed. Their new functionality is similar to their previous behavior, but their names have changed to clarify that MIP starts exist as objects separate from the solution pool and that a user can manage more than one MIP start at the same time. The new methods and routines make it possible for a user to add, modify, and query **multiple** MIP starts.

Concert Technology deprecated methods for MIP start

Deprecated method	New recommended method
C++ API	
<code>IloCplex::readMIPStart</code>	<code>IloCplex::readMIPStarts</code>
<code>IloCplex::writeMIPStart</code>	<code>IloCplex::writeMIPStarts</code>
Java API	
<code>IloCplex.readMIPStart</code>	<code>IloCplex.readMIPStarts</code>
<code>IloCplex.writeMIPStart</code>	<code>IloCplex.writeMIPStarts</code>
.NET API	
<code>Cplex.ReadMIPStart</code>	<code>Cplex.ReadMIPStarts</code>
<code>Cplex.WriteMIPStart</code>	<code>Cplex.WriteMIPStarts</code>

Callable Library deprecated routines for MIP start

Deprecated routine	New recommended routine
<code>CPXgetsolnpoolmipstart</code>	<code>CPXgetmipstarts</code>
<code>CPXgetmipstart</code>	<code>CPXgetmipstarts</code>
<code>CPXmstwrite</code>	<code>CPXwritemipstarts</code>
<code>CPXmstwritesolnpool</code>	<code>CPXwritemipstarts</code>
<code>CPXmstwritesolnpoolall</code>	<code>CPXwritemipstarts</code>
<code>CPXgetsolnpoolnummipstarts</code>	<code>CPXgetnummipstarts</code>
<code>CPXchgmpstart</code>	<code>CPXchgmpstarts</code>
<code>CPXcopympstart</code>	<code>CPXdelmipstarts</code> followed by <code>CPXaddmipstarts</code>

Deprecated routine	New recommended routine
CPXreadcopymipstart	CPXreadcopymipstarts

Continuous variables now in MIP starts

Methods and routines to access MIP starts now return values for **continuous variables** as well as discrete variables.

◆ In Concert Technology

- In the C++ API, `IloCplex::getMIPStart`
- In the Java API, `IloCplex.getMIPStart`
- In the .NET API, `Cplex.GetMIPStart`

◆ In the Callable Library, `CPXgetmipstarts`

The values for continuous variables may be useful when CPLEX solves a problem with identical constraints but a different objective.

If the user is interested in only the values of discrete variables (that is, the **previous behavior**), then the user should query the variable types, isolate the discrete variables (for example, in a user-defined array), and query values of only those discrete variables.

MIP starts independent of the solution pool

Outlines changes due to multiple MIP starts being independent of the solution pool.

In this section

Solution pool: existing solutions eligible for replacement

Announces change in solution pool with respect to existing solutions.

Solution pool: changed indices

Warns about a change in indices of solutions in the solution pool.

Solution pool, multiple MIP starts, and the incumbent

Explains impact of multiple MIP starts on the incumbent and the solution pool.

Solution pool: examining solutions

Explains how to enumerate all solutions in the solution pool, in view of changed indices.

Solution pool: existing solutions eligible for replacement

As noted in *Impact of multiple MIP starts*, ILOG CPLEX 11.2 supports multiple MIP starts **independent of the solution pool**. Consequently, this note highlights conversion issues resulting from this new feature, with respect to the solution pool. For more information about the new feature itself, see also *Multiple MIP starts* in these release notes.

Previously, when MIP optimization or populate (the procedure for accumulating solutions in the solution pool) were invoked repeatedly, CPLEX left untouched any solutions already in the solution pool.

This behavior has been changed: Now, CPLEX may replace solutions in the solution pool if the pool is at its capacity and CPLEX finds new solutions satisfying the replacement criteria.

If a user wants to keep all solutions produced through all calls to MIP optimization or populate, then the user must query the solution pool before calling MIP optimization or populate again and store the solutions in user-defined arrays.

Solution pool: changed indices

Because of changes in the solution pool to handle MIP starts as explicit, separate objects and because of changes to support multiple MIP starts, the indexing of solutions in the solution pool has changed. An application relying on indexing of solutions in the solution pool may continue to compile correctly but give unexpected results when running.

Solution pool, multiple MIP starts, and the incumbent

As long as the capacity of the solution pool is one or more, the incumbent now always appears as the solution with index 0 (zero) after MIP optimization or populate, even if the incumbent had been deleted from the pool prior to the invocation of MIP optimization or populate.

A copy of the incumbent solution is always added to the pool, as long as the pool capacity is at least one, regardless of its evaluation with respect to any filters and regardless of the replacement criterion governing the solution pool. (Previously, the incumbent might have been missing from the pool due to filtering or due to the replacement strategy.) This copy of the incumbent solution will be the first member of the solution pool, that is, the solution with index 0 (zero).

Previously, the incumbent was available in two ways from the solution pool: as the member with index -1 (minus one) or possibly as an arbitrarily numbered member of the solution pool.

Tip: If your application accesses the incumbent by the index -1 (minus one), it will continue to work, but best practice suggests changing an application to use the symbolic value referring to the incumbent.

In ILOG CPLEX 11.2, the incumbent is available through query methods or routines using a special symbolic value.

- ◆ In Concert Technology, use the symbolic value `IncumbentId` as an argument to a method.
 - In the C++ API, use `IloCplex::IncumbentId` as the argument to specify the solution index in methods such as `IloCplex::getValues`. For a complete list of methods that should use `IloCplex::IncumbentId` to access the incumbent, see *Accessing the incumbent in the C++ API*.
 - In the Java API, use `IloCplex.IncumbentId` as the argument to specify the solution index in methods such as `IloCplex.getValues`. For a complete list of methods that should use `IloCplex::IncumbentId` to access the incumbent, see *Accessing the incumbent in the Java API*.
 - In the .NET API, use `Cplex.IncumbentId` as the argument to specify the solution index in methods such as `Cplex.GetValues`. For a complete list of methods that should use `IloCplex::IncumbentId` to access the incumbent, see *Accessing the incumbent in the .NET API*.
- ◆ In the Callable Library, use the symbolic value `CPX_INCUMBENT_ID` as an argument to specify the solution index in routines such as `CPXgetsolnpoolx`. For a complete list of routines that should use `CPX_INCUMBENT_ID`, see *Accessing the incumbent in the Callable Library*.

◆ In the Interactive Optimizer, for a solution in SOL file format, use one of these commands, where `filename` is the name you supply of the file to write to:

- `write filename.sol`
- `write filename.sol 0`
- `write filename.sol incumbent`

Also in the Interactive Optimizer, if MIP start 1 (one) has been deleted or changed, it no longer corresponds to the incumbent. As long as there has been no change to the MIP starts since the previous populate or optimization call, you can write the incumbent as a MIP start to a file in MST format like this:

- `write filename.mst 1`

The incumbent remains a member of the solution pool as member 0 (zero) on return from a MIP optimization or populate as long as the capacity of the solution pool is positive.

The incumbent remains available by means of other query methods, such as `getValues` or routines such as `CPXgetx`, without an index specified from the solution pool. That is, the incumbent remains accessible apart from the solution pool.

Solution pool: examining solutions

If a user wants to examine all the solutions available in the solution pool, the application should now loop from 0 (zero) to N-1 (that is, one less than the number of solutions in the pool).

Tip: If your application loops through the solutions of the solution pool from -1 (minus one), then you need to change your application so that the loop starts at 0 (zero).

To learn the number of solutions in the pool for use in such a loop, ILOG CPLEX 11.2 offers the following methods or routines.

- ◆ In Concert Technology,
 - In the C++ API, use the method `IloCplex::getSolnPoolNsolns`.
 - In the Java API, use the method `IloCplex.getSolnPoolNsolns`.
 - In the .NET API, use the method `Cplex.GetSolnPoolNsolns`.
- ◆ In the Callable Library, `CPXgetsolnpoolnumsolns`.

Previously, an application had to loop from -1 (minus one) to one less than the number of solutions in the pool. However, in some circumstances, the incumbent might possibly be processed twice in such a loop. Improvements in ILOG CPLEX 11.2 now prevent the possibility of duplicates.

Change in XML schema

The XML element formerly known as `<CPLEXSolutionPool>` has changed. The element is now `<CPLEXSolutions>`, a more generic name supporting solutions, multiple MIP starts, and members of the solution pool. This XML element appears in the files `solution.xml` and `solution.xsd` for XML serialization of CPLEX solutions.

If your XML parser recognized the former element, you need to change your parser to recognize the new name, `<CPLEXSolutions>`.

Lower and upper cutoff tolerances

The documentation of the lower and upper cutoff tolerances has been revised to reflect their behavior more accurately. For more detail, see *lower cutoff* and *upper cutoff* in the *ILOG CPLEX Parameter Reference Manual*.

- ◆ For Concert Technology, see `CutLo` and `CutUp`.
- ◆ For the Callable Library, see `CPX_PARAM_CUTLO` and `CPX_PARAM_CUTUP`.
- ◆ For the Interactive Optimizer, see `mip tolerances lowercutoff` and `mip tolerances uppercutoff`.

User cuts may be purged

ILOG CPLEX 11.2 makes it possible for you to specify to CPLEX to purge global cuts that you add to the model by means of a cut callback; that is, you can authorize CPLEX to remove global user cuts added to the model using cut callbacks. If you have specified that a cut you added is **purgeable**, then CPLEX may determine at any time that the cut is no longer useful and then purge it. By purging cuts, CPLEX eliminates those which are redundant with other aspects of the model, saves computation of the unnecessary cuts, and thus may improve performance in some situations.

- ◆ In Concert Technology, new methods with an additional argument allow you to specify to CPLEX whether a cut you added to the model is can be purged when CPLEX decides to do so.
 - In the C++ API, see the method `IloCplex::UserCutCallbackI::add`.
 - In the Java API, see the method `IloCplex.UserCutCallback.add`.
 - In the .NET API, see the method `Cplex.UserCutCallback.Add`.
- ◆ In the Callable Library, the routine `CPXcutcallbackadd` now has an additional argument with which you can tell CPLEX whether a cut you added to the model can be purged when CPLEX decides to do so.

In order to take advantage of this new feature, you must edit your application to use the new argument of the relevant method or routine, recompile, and relink.

Conversion notes for users of MS Windows

In previous releases, only wall clock time was available for measuring computation time in ILOG CPLEX applications on Microsoft Windows. Now, however, the parameter to select *clock type for computation time* has been extended on Microsoft platforms to support both **CPU time** and **wall clock time** (as before).

At the default setting of this parameter, ILOG CPLEX 11.2 decides automatically whether to use CPU time or wall clock time to measure computation both to report performance and to terminate optimization when the user has set a time limit. Conventionally, ILOG CPLEX chooses wall clock time when other parameters invoke parallel optimization and chooses CPU time when other parameters enforce sequential (not parallel) optimization.

For documentation of this parameter, see *clock type for computation time* in the *ILOG CPLEX Parameters Reference Manual*.

- ◆ `ClockType` in Concert Technology
- ◆ `CPX_PARAM_CLOCKTYPE` in the Callable Library
- ◆ `clocktype` in the Interactive Optimizer

Conversion notes for users of Concert Technology

Describes changes of interest to those who use Concert Technology. A new version of Concert Technology (Concert Technology 2.7) accompanies this release of ILOG CPLEX 11.2. This new version requires you to recompile and link your Concert Technology applications.

In this section

Conversion notes for users of C++ API

Describes changes of interest to those who use the C++ API of CPLEX.

Conversion notes for users of Java API

Describes changes of interest to those who use the Java API of CPLEX.

Conversion notes for users of .NET API

Describes changes of interest to those who use the .NET API of CPLEX.

Conversion notes for users of C++ API

These conversion notes apply to C++ applications using ILOG CPLEX 11.2.

- ◆ *Accessing the incumbent in the C++ API*
- ◆ *IloCplex::getRay returns sparse array*

Accessing the incumbent in the C++ API

As explained in *Solution pool, multiple MIP starts, and the incumbent*, changes to support multiple MIP starts and changes in the numbering or indexing of solutions in the solution pool imply that Concert Technology methods accessing the incumbent should use the symbolic value `IloCplex::IncumbentId`.

Tip: If your application uses any of the following methods, you need to change it.

The following methods of the C++ API of Concert Technology may be affected by this change in ILOG CPLEX 11.2 and must use `IloCplex::IncumbentId` to access the incumbent:

- ◆ `getSlack`
- ◆ `getSlacks`
- ◆ `getQuality`
- ◆ `getObjValue`
- ◆ `writeMIPStart`
- ◆ `solveFixed`

IloCplex::getRay returns sparse array

The method `IloCplex::getRay` now returns the unbounded direction (also known as a ray) as a **sparse array** instead of a dense array. Consequently, it provides only the variables and values associated with the **nonzero** elements of the ray.

Conversion notes for users of Java API

These conversion notes apply to Java applications using ILOG CPLEX 11.2.

- ◆ *Accessing the incumbent in the Java API*
- ◆ *IloNumVar now implements IloAddable*

Accessing the incumbent in the Java API

As explained in *Solution pool, multiple MIP starts, and the incumbent*, changes to support multiple MIP starts and changes in the numbering or indexing of solutions in the solution pool imply that Concert Technology methods accessing the incumbent should use the symbolic value `IloCplex.IncumbentId`.

Tip: If your application uses any of the following methods, you need to change it.

The following methods of the Java API of Concert Technology may be affected by this change in ILOG CPLEX 11.2 and must use `IloCplex.IncumbentId` to access the incumbent:

- ◆ `getSlack`
- ◆ `getSlacks`
- ◆ `getQuality`
- ◆ `getObjValue`
- ◆ `writeMIPStart`
- ◆ `solveFixed`

IloNumVar now implements IloAddable

The Java interface `IloNumVar` now implements the interface `IloAddable`. The practical effect of this change is that a user can now add a variable (an instance of `IloNumVar`) to a model directly.

Previously, a variable was added tacitly when it participated in a constraint and the constraint was added to a model. As a consequence of that previous design, it was possible for a call of `getValue` to raise an exception for a variable that did not participate in a constraint and had consequently not been added to the model or for a variable participating in a constraint that had not yet been added to the model.

In the new design, with variables as addable objects, it is now possible to add a variable to the model directly, even if the variable is not used in a constraint nor in the objective function. An exception will no longer be raised when `getValue` queries such a variable, for example.

Conversion notes for users of .NET API

These conversion notes apply to .NET applications using ILOG CPLEX 11.2.

- ◆ *Accessing the incumbent in the .NET API*
- ◆ *INumVar now implements IAddable*

Accessing the incumbent in the .NET API

As explained in *Solution pool, multiple MIP starts, and the incumbent*, changes to support multiple MIP starts and changes in the numbering or indexing of solutions in the solution pool imply that Concert Technology methods accessing the incumbent should use the symbolic value `Cplex.IncumbentId`.

Tip: If your application uses any of the following methods, you need to change it.

The following methods of the .NET API of Concert Technology may be affected by this change in ILOG CPLEX 11.2 and must use `Cplex.IncumbentId` to access the incumbent:

- ◆ `GetSlack`
- ◆ `GetSlacks`
- ◆ `GetQuality`
- ◆ `GetObjValue`
- ◆ `WriteMIPStart`
- ◆ `SolveFixed`

INumVar now implements IAddable

The .NET interface `INumVar` now implements the interface `IAddable`. The practical effect of this change is that a user can now add a variable (an instance of `INumVar`) to a model directly.

Previously, a variable was added tacitly when it participated in a constraint and the constraint was added to a model. As a consequence of that previous design, it was possible for a call of `GetValue` to raise an exception for a variable that did not participate in a constraint and had consequently not been added to the model or for a variable participating in a constraint that had not yet been added to the model.

In the new design, with variables as addable objects, it is now possible to add a variable to the model directly, even if the variable is not used in a constraint nor in the objective function. An exception will no longer be raised when `GetValue` queries such a variable, for example.

Conversion notes for users of the Callable Library

These conversion notes apply to Callable Library (C API) applications using ILOG CPLEX 11.2.

- ◆ *Routines removed from this release*
- ◆ *Accessing the incumbent in the Callable Library*
- ◆ *CPXgetconflict accesses correct number of variables*

Routines removed from this release

These routines were deprecated in a previous release and have been removed from this release of the Callable Library (C API):

- ◆ CPXlpread
- ◆ CPXlpmread
- ◆ CPXlpqread
- ◆ CPXmpsread
- ◆ CPXmpsqread
- ◆ CPXsavread
- ◆ CPXsavmread
- ◆ CPXsavqread
- ◆ CPXmbaseread

ILOG recommends that you use CPXreadcopyprob instead, followed by a query of the CPXLP object for your target data by means of such query routines as CPXgetrows, CPXgetobjsense, CPXgetsense. Examples demonstrating the use of CPXreadcopyprob are available in the samples distributed with this version of the product in yourCPLEXhome/examples.

Accessing the incumbent in the Callable Library

As explained in *Solution pool, multiple MIP starts, and the incumbent*, changes to support multiple MIP starts and changes in the numbering or indexing of solutions in the solution pool imply that Callable Library routines accessing the incumbent should use the symbolic value CPX_INCUMBENT_ID.

Tip: If your application uses one of the following routines, you need to change it.

The following routines of the Callable Library may be affected by this change in ILOG CPLEX 11.2 and must use `CPX_INCUMBENT_ID` to access the incumbent:

- ◆ `CPXgetsolnpoolobjval`
- ◆ `CPXgetsolnpoolslack`
- ◆ `CPXgetsolnpoolqconstrslack`
- ◆ `CPXgetsolnpooldblquality`
- ◆ `CPXgetsolnpoolintlquality`
- ◆ `CPXgetsolnpoolsolnname`
- ◆ `CPXsolwritesolnpool`
- ◆ `CPXgetsolnpoolmipstart` (This routine is **deprecated**; use `CPXgetmipstarts` instead.)
- ◆ `CPXmstwritesolnpool` (This routine is **deprecated**; use `CPXwritemipstarts` instead.)
- ◆ `CPXchgprobtypesolnpool`

CPXgetconflict accesses correct number of variables

`CPXgetconflict` now lists a given variable only once in the results that it accesses from `CPXrefineconflict`.

Previously, it was possible for `CPXgetconflict` to list a variable twice: once with the status `CPX_CONFLICT_POSSIBLE_LB` when the lower bound of the variable might possibly participate in a conflict, and again with the status `CPX_CONFLICT_POSSIBLE_UB` when the upper bound of the variable might possibly participate in the conflict. However, if one bound of a variable (either the lower or upper bound) has been **proven** to participate in a conflict, then the other bound can **not** be part of the same conflict.

`CPXgetconflict` now takes into account this fact about the bounds of a variable. Consequently, the status of a variable is:

- ◆ `CPX_CONFLICT_POSSIBLE_LB` strictly when **only** the lower bound of the variable may possibly be in the conflict;
- ◆ `CPX_CONFLICT_POSSIBLE_UB` strictly when **only** the upper bound of the variable may possibly be in the conflict;

- ◆ `CPX_CONFLICT_POSSIBLE_MEMBER` strictly when **both** bounds of a variable may possibly be in the conflict.

New features for all users

Announces new features in ILOG CPLEX 11.2.

In this section

New parameters

Summarizes new parameters available in this release.

New error codes

Lists new error codes in this release.

Solving more SOCP, QCP, and MIQCP

Describes enlarged scope of Second Order Cone Programs solved by CPLEX.

Multiple MIP starts

Introduces new features for starting a mixed integer program from advanced solutions (MIP starts).

Write level for SOL, MST file formats

Describes a new parameter to support write level for MIP starts.

Refining a conflict in a MIP start

Describes how a MIP start may be used with the conflict refiner.

Accessing MIP relative gap

Introduces new methods, routines to access the relative gap in a MIP.

New ways to measure computational time

Announces new methods, routines to measure computational time

New parameters

New parameters in ILOG CPLEX 11.2

Reference Manual	Concert Technology	Callable Library	Interactive Optimizer
<i>absolute MIP gap before starting to polish a feasible solution</i>	PolishAfterEpAGap	CPX_PARAM_POLISHAFTEREPAGAP	mip polishafter absmipgap
<i>relative MIP gap before starting to polish a feasible solution</i>	PolishAfterEpGap	CPX_PARAM_POLISHAFTEREPGAP	mip polishafter mipgap
<i>MIP integer solutions to find before starting to polish a feasible solution</i>	PolishAfterIntSol	CPX_PARAM_POLISHAFTERINTSOL	mip polishafter solutions
<i>nodes to process before starting to polish a feasible solution</i>	PolishAfterNode	CPX_PARAM_POLISHAFTERNODE	mip polishafter nodes
<i>time before starting to polish a feasible solution</i>	PolishAfterTime	CPX_PARAM_POLISHAFTERTIME	mip polishafter time
<i>write level for MST, SOL files</i>	WriteLevel	CPX_PARAM_WRITELEVEL	write level i

New error codes

Error code	Symbolic name	Explanation
1807	CPXERR_PARAM_INCOMPATIBLE	Incompatible parameters cannot be used together. In particular, the deprecated parameter <code>PolishTime</code> or <code>CPX_PARAM_POLISHTIME</code> cannot be used with any of the parameters with <i>polishafter</i> in their name, such as <code>PolishAfter___</code> or <code>CPX_PARAM_POLISH___</code> .

Solving more SOCP, QCP, and MIQCP

ILOG CPLEX 11.2 solves a larger class of models with **quadratic** terms among the constraints.

In previous releases, CPLEX solved quadratically constrained programs (QCP) and mixed integer quadratically constrained programs (MIQCP) **only** if they satisfied at least one of these conditions:

```
x'Qx <= e where e is a linear expression
```

```
x'x <= y2 where y >= 0
```

Now CPLEX solves a larger class of problems with quadratic terms among the constraints. CPLEX transforms quadratic constraints automatically into second order cone constraints of these types:

- ◆ $x'Qx \leq e$ where e is a linear expression and Q is PSD. Previous versions of CPLEX handled this type.
- ◆ $x'Qx \leq y^2$ where $y \geq 0$ and Q is PSD. Previous versions of CPLEX handled only a subset of this type, specifically, the subset where Q is the identity matrix. Now CPLEX automatically translates all quadratic constraints of this type.
- ◆ $x'Qx \leq yz$ where $y \geq 0$, $z \geq 0$, and Q is PSD, new in ILOG CPLEX 11.2.

Multiple MIP starts

ILOG CPLEX 11.2 supports **multiple** MIP starts; that is, a user may maintain more than one starting solution with values for continuous and discrete variables for CPLEX to use as an advanced starting point. MIP starts are also known as advanced starts or warm starts.

Tip: The new features supporting multiple MIP starts described in these release notes apply only to mixed integer programs (MIPs); these new features do not change the behavior of advanced starts for continuous models.

MIP starts independent of the solution pool

Previously, an implicit MIP start was associated with each solution in the solution pool. Accordingly, if a solution was deleted from the solution pool, the associated MIP start was also deleted. Moreover, a user could not modify those implicit MIP starts belonging to the solution pool.

Now, however, the MIP starts are explicit; that is, MIP starts corresponding to solutions in the solution pool are separate, independent objects. Both MIP optimization and populate now produce a separate, explicit MIP start corresponding to each solution in the solution pool.

Explicit MIP starts enable a user to manage MIP starts independently of the solution pool.

For example, a user may **modify** the MIP start created from a solution of the solution pool. In fact, a user may delete both the solutions in the solution pool and the separate, independent MIP starts.

A user may also **modify** any of the MIP starts, both the MIP start derived from the incumbent and the MIP starts derived from the solutions in the solution pool. The next invocation of MIP optimization or populate (the procedure for accumulating solutions in the solution pool) processes all MIP starts.

In short, a user may **query**, **modify**, or **delete** any of the multiple MIP starts.

New methods, routines for multiple MIP starts

Action	Concert Technology	Callable Library
add MIP starts to a model	addMIPStart	CPXaddmipstarts
modify MIP starts	changeMIPStart	CPXchgmmipstarts
query MIP starts	getMIPStart	CPXgetmipstarts
query number of MIP starts	getNMIPStarts	CPXgetnumipstarts
delete MIP starts	deleteMIPStarts	CPXdelmipstarts CPXdelsetmipstarts

Reading, writing multiple MIP starts

A user may also write these MIP starts to a file, for example, in **MST** format. In fact, the user may add multiple MIP starts to such a file or read multiple MIP starts from a single file.

Action	Concert Technology	Callable Library
write MIP starts to a file	writeMIPStarts	CPXwritemipstarts
read MIP starts from a file	readMIPStarts	CPXreadcopymipstarts

Naming of multiple MIP starts

The user may name MIP starts and refer to them by name in order to facilitate management of multiple MIP starts.

Action	Concert Technology	Callable Library
access the name of a MIP start	getMIPStartName	CPXgetmipstartname

Indexing of multiple MIP starts

As with other CPLEX objects, such as variables or constraints, when a user deletes a MIP start, CPLEX automatically updates the index of the following MIP starts. There are new methods and routines for the user to access or to recalculate indices after a deletion.

Action	Concert Technology	Callable Library
access the index of a MIP start	getMIPStartIndex	CPXgetmipstartindex

Multiple MIP starts and effort level

Users may want CPLEX to process multiple MIP starts differently, expending more effort on some than on others. Moreover, a user may want to limit the effort CPLEX applies to MIP starts when it transforms each MIP start into a feasible solution, especially if there are many of them. In that context, the user may specify a **level of effort** that CPLEX should expend for each MIP start to transform it into a feasible solution. The user specifies the level of effort as an argument to the method or routine that adds a MIP start to a model or that modifies a MIP start.

CPLEX applies the effort level specified by the user. If a MIP start does not contain enough information to be processed with the effort level specified by the user, then CPLEX will not process this MIP start. For example, if the MIP start does not specify values for all variables

but the user has specified an effort level to check feasibility, CPLEX does not process the MIP start.

Here are the levels of effort and their effect:

- ◆ Level 1: CPLEX checks **feasibility** of the corresponding MIP start. This level requires the user to provide values for **all** variables in the problem, both discrete and continuous. If any values are missing, CPLEX does not process this MIP start.
- ◆ Level 2: CPLEX solves the **fixed LP** problem specified by the MIP start. This effort level requires the user to provide values for all **discrete** variables. If values for any discrete variables are missing, CPLEX does not process the MIP start. CPLEX uses the values of continuous variables at this effort level only in the initial phase when it attempts to determine values for unspecified discrete variables.
- ◆ Level 3: CPLEX solves a subMIP. The user must specify the value of at least one discrete variable at this effort level. CPLEX uses the values of continuous variables at this effort level only in the initial phase when it attempts to determine values for unspecified discrete variables.
- ◆ Level 4: CPLEX attempts to repair the MIP start if it is infeasible, according to the parameter that sets the *frequency to try to repair infeasible MIP start* (RepairTries, CPX_PARAM_REPAIRTRIES). The user must specify the value of at least one discrete variable at this effort level, too.

A user may specify a different level of effort for each MIP start, for example, differing levels of effort for the incumbent, for a MIP start corresponding to a solution in the solution pool, for a MIP start supplied by the user. By **default**, CPLEX expends effort at level 4 for the first MIP start (which will be the incumbent after a MIP optimization or a call to populate) and at level 1 (one) for all other MIP starts. The user may change that level of effort.

The reference manuals of the APIs document the symbolic names of the MIP start effort levels.

- ◆ In Concert Technology:
 - in the C++ API, see the enumeration `IloCplex::MIPStartEffort`.
 - in the Java API, see the enumeration `IloCplex.MIPStartEffort`.
 - in the .NET API, see the members of `Cplex.MIPStartEffort`.
- ◆ In the Callable Library, see the documentation of the routine to add MIP starts, `CPXaddmipstarts`, and the routine to change MIP starts, `CPXchgmmipstarts`.

Multiple MIP starts and MST files

A file in MST format (that is, a formatted file with the extension `.mst`) may contain many MIP starts. Previously, CPLEX processed only the first MIP start from such a file. Now CPLEX can process all MIP starts read from a formatted MST file.

Write level for SOL, MST file formats

A new parameter (`WriteLevel`, `CPX_PARAM_WRITELEVEL`) controls which information CPLEX writes to SOL and MST files (that is, formatted files with the extension `.sol` for solutions or `.mst` for MIP starts). CPLEX records the write level at which it created a file in that file, so that the file can be read back accurately later.

At its automatic setting, level 0 (zero), CPLEX maintains its **previous behavior**; that is, CPLEX writes all variables and their values to SOL files; CPLEX writes all discrete variables and their values to MST files. This automatic setting at level 0 (zero) is the **default**.

Values of the write level parameter

Level	Concert Technology / Callable Library	Meaning
0	Auto CPX_WRITELEVEL_AUTO	Automatic: Let CPLEX decide. (default)
1	AllVars CPX_WRITELEVEL_ALLVARS	Write all values, both discrete and continuous variables.
2	DiscreteVars CPX_WRITELEVEL_DISCRETEVARS	Write values for discrete variables only.
3	NonzeroVars CPX_WRITELEVEL_NONZEROVARS	Write values for all variables with a nonzero value.
4	NonzeroDiscreteVars CPX_WRITELEVEL_NONZERODISCRETEVARS	Write values for nonzero discrete variables only.

Levels 3 and 4 reduce the size of files, of course. However, this reduced file size implies some restrictions, as documented in *write level for MST, SOL files* of the *ILOG CPLEX Parameter Reference Manual*.

With respect to levels 3 and 4, where **nonzero** values are significant, CPLEX considers a value nonzero if the absolute value is strictly less than $1e-16$. In the case of **SOL** files, CPLEX applies this test to **primal** and **dual variable** values, that is, both `x` and `pi` variable values. In the case of **MST** files, CPLEX applies this test only to `x` values.

Refining a conflict in a MIP start

The **conflict refiner** can now accept a MIP start. This feature may aid a user in debugging a MIP start that CPLEX rejects or in debugging the model itself, if the user knows that the pairs of variables and values provided as a MIP start should be a feasible solution of the problem. That is, a user can use the conflict refiner to analyze a rejected MIP start and may then be able to repair the MIP start appropriately.

To use the conflict refiner to debug a rejected MIP start, follow these steps:

1. Add the MIP start to the CPLEX object.
2. Call a special version of the conflict refiner, specifying which MIP start to consider:
 - ◆ In Concert Technology
 - In the C++ API, use `IloCplex::refineMIPStartConflict`.
 - In the Java API, use `IloCplex.refineMIPStartConflict`.
 - In the .NET API, use `Cplex.RefineMIPStartConflict`.
 - ◆ In the Callable Library, use `CPXrefinemipstartconflict` or `CPXrefinemipstartconflictext`.
 - ◆ In the Interactive Optimizer, use the command `conflict i` where `i` specifies the index of the MIP start.

When a MIP start is added to the current model, an effort level may be specified to tell CPLEX how much effort to expend in transforming the MIP start into a feasible solution. The conflict refiner respects effort levels **except** level 1 (one): check feasibility. It does **not** check feasibility of a MIP start but instead applies effort level 2, to solve a fixed MIP.

Accessing MIP relative gap

These new methods and routines access the **relative gap** in a MIP, even from **callbacks**.

◆ In Concert Technology, use these methods:

- In the C++ API, `IloCplex::getMIPRelativeGap` and `MIPInfoCallbackI::getMIPRelativeGap`
- In the Java API, `IloCplex.getMIPRelativeGap` and `MIPInfoCallback.getMIPRelativeGap`
- In the .NET API, `Cplex.GetMIPRelativeGap` and `MIPInfoCallback.GetMIPRelativeGap`

◆ In the Callable Library, use the routine `CPXgetmiprelgap` and from callbacks, use `.CPXgetcallbackinfo` with the symbolic value `CPX_CALLBACK_INFO_MIP_REL_GAP` as its argument `whichinfo`.

New ways to measure computational time

These new methods, properties, and routines provide a **time stamp** to enable users to measure computational time. For example, an application can invoke one of these methods or routines at the beginning and end of an operation; then compare the two time stamps to compute elapsed time in seconds.

- ◆ In Concert Technology
 - In the C++ API, `IloCplex::getCplexTime` returns a time stamp that applications may use to calculate elapsed time in seconds.
 - In the Java API, `IloCplex.getCplexTime` returns a time stamp that applications may use to calculate elapsed time in seconds.
 - In the .NET API, the property `Cplex.CplexTime` accesses a time stamp that applications may use to calculate elapsed time in seconds.
- ◆ In the Callable Library, `CPXgetTime` returns a time stamp that applications may use to calculate elapsed time in seconds.
- ◆ This feature was already available in the Interactive Optimizer.

In addition, other new methods and routines return a time stamp specifying when a time limit will occur; this time stamp may be used in calculations with the time stamp returned by the previous methods or routines to compute remaining time in seconds from **callbacks**.

- ◆ In Concert Technology
 - In the C++ API, `CallbackI::getEndTime` returns a time stamp specifying when a time limit will occur; this time stamp may be used in calculations with the time stamp returned by `getTime` to compute remaining time in seconds.
 - In the Java API, `CpxCallback.getEndTime` returns a time stamp specifying when a time limit will occur; this time stamp may be used in calculations with the time stamp returned by `getTime` to compute remaining time in seconds.
 - In the .NET API, the property `Cplex.Callback.EndTime` accesses a time stamp specifying when a time limit will occur; this time stamp may be used in calculations with the time stamp accessed by `CplexTime` to compute remaining time in seconds.
- ◆ In the Callable Library, `CPX_CALLBACK_INFO_ENDTIME` is a new value that `CPXgetcallbackinfo` can supply in its argument `whichinfo`. That value is a time stamp of the point in time when the optimization will terminate if optimization does not finish before that point. In other words, this symbol is useful in measuring time through information callbacks.

For a sample of these new features, see these examples among those distributed with the product in `yourCPLEXinstallation/examples`:

- ◆ `ilomipex4.cpp` in C++ in Concert Technology
- ◆ `MIPex4.java` in Java in Concert Technology
- ◆ `MIPex4.cs` in C#.NET in Concert Technology
- ◆ `MIPex4.vb` in Visual Basic.NET in Concert Technology
- ◆ `mipex4.c` in C in the Callable Library

ILOG CPLEX 11.1 Release Notes

Thank you for installing ILOG CPLEX 11.1. These release notes highlight improvements in ILOG CPLEX 11.1. Please review these notes before using ILOG CPLEX 11.1.

In this section

Welcome to ILOG CPLEX 11.1

Describes general outlines of ILOG CPLEX 11.1.

Conversion notes for all users

Describes changes important to all customers.

Conversion notes for Concert Technology users

Describes changes of interest to customers who use Concert Technology.

Conversion notes for Callable Library users

Describes changes of interest to customers who use the Callable Library (C API).

Welcome to ILOG CPLEX 11.1

Describes general outlines of ILOG CPLEX 11.1.

In this section

Announcements

Highlights news about ILOG CPLEX 11.1.

Port changes

Summarizes deprecated or removed ports and recommends replacements.

Announcements

If you are already using ILOG CPLEX 11.0, there may be little or no need to change your application in order to take advantage of improvements in ILOG CPLEX 11.1 because there are no new API features in this release. Just **recompile** and **link** your application with ILOG CPLEX 11.1 component libraries.

Please consider the following descriptions of changes and improvements to see whether they may apply to your particular situation nevertheless.

Port changes

For a complete list of machine types and library formats (including version numbers of compilers and JDKs) see the file `yourCPLExhome/mptable.html` in the distribution of the product. That file contains more detailed descriptions of available ports than do these highlights.

New ports

The following new ports are available with this release:

- ◆ New ports, suitable for **x86** and **x86-64** processors, and built on the Debian 4 distribution of the **GNU/Linux** operating system, are available in this release. These new ports are compatible with the Red Hat Enterprise Linux 5 (**RHEL5**) and Suse Linux Enterprise Server 10 (**SLES10**).
- ◆ New ports supporting **Sun UltraSPARC** processors and **Sun Studio 9** or higher compilers and development environment are available for 32-bit and for 64-bit architectures.
- ◆ A new port for **Itanium HP-UX** architecture, compatible with the operating system **HP-UX 11i**, based on the **aC++ 6.17** compiler, is available.
- ◆ A new port compatible with Microsoft **Visual Studio 2008** is available.

Summary of new ports

Port name	Platform
x86_debian4.0_4.1 x86-64_debian4.0_4.1	Debian 4 GNU/Linux Red Hat Enterprise Linux 5 (RHEL5) Suse Linux Enterprise Server 10 (SLES10)
ultrasparc32_9_9 ultrasparc64_9_9	Sun Solaris 9 or higher operating system Sun Studio 9 or higher compiler
ia64_hpux11_6.17	Itanium processor family HP-UX 11i operating system HP-aC++ 6.17 or higher compiler
x86_windows_vs2008 x64_windows_vs2008	X86 or X64 machine architecture Microsoft Visual Studio 2008 compiler .NET Framework 2.0 or higher

Deprecated and removed ports

Summary of deprecated and removed ports

Port name	Platform	Status	Recommended replacement
ultrasparc32_9_8	Sun Studio 8, 32-bit	deprecated: no longer available in the next release	ultrasparc32_9_9 for Sun Studio 9 and higher
ultrasparc64_9_8	Sun Studio 8, 64-bit	deprecated: no longer available in the next release	ultrasparc64_9_9 for Sun Studio 9 and higher
power32_aix5.2_7.0	IBM Power AIX 5.2 IBM XL C/C++ 7.0 32-bit	deprecated: no longer available in the next release	A new port, compatible with IBM Power AIX 5.3 or higher, will be available in the next release.
power64_aix5.2_7.0	IBM Power AIX 5.2 IBM XL C/C++ 7.0 64-bit	deprecated: no longer available in the next release	A new port, compatible with IBM Power AIX 5.3 or higher, will be available in the next release.
x86_rhel4.0_3.4 x86-64_rhel4.0_3.4	Red Hat Enterprise Linux 4	deprecated and removed: no longer available now	x86_debian4.0_4.1 or x86-64_debian4.0_4.1 compatible with Debian 4, Red Hat Enterprise Linux 5, or Suse Linux Enterprise Server 10
x86_.net2003_7.1 ia64_.net2003_7.1	Microsoft Visual Studio 2003	removed: no longer available now	x86_windows_vs2008 or x64_windows_vs2008

Sun Studio 8

The ports named ultrasparc32_9_8 and ultrasparc64_9_8, compatible with Sun Studio 8 and higher, are **deprecated** and will not be supported in versions after CPLEX 11.1.

The **recommended replacements** are the new ports ultrasparc32_9_9 and ultrasparc64_9_9, compatible with Sun Studio 9 and higher. The Sun Studio 9 ports are newly available in CPLEX 11.1.

IBM Power AIX 5

The ports named power32_aix5.2_7.0 and power64_aix5.2_7.0 are **deprecated**.

IBM announced plans to stop support for AIX 5.2 in the next twelve months. The **recommended replacement** is migration to AIX 5.3 or a higher version, also supported by this port.

Red Hat Enterprise Linux 4

Ports based on Red Hat Enterprise Linux version 4 (that is, those with `_rhel4_` in their name) are **deprecated** and have been **removed**.

The recommended replacements are named `x86_debian4.0_4.1` and `x86_64_debian4.0_4.1`. These ports are compatible with Red Hat Enterprise Linux version 5, Suse Linux Enterprise Server version 10, and Debian version 4.

Microsoft Visual Studio 2003

The ports based on Visual Studio 2003 have been **removed**. **Recommended replacements** are ports built on Microsoft Visual Studio 2005 or Microsoft Visual Studio 2008.

Conversion notes for all users

Describes changes important to all customers.

In this section

Conversion overview

Describes the migration path from earlier versions to this version of the product.

Solution polishing and MIP starts

Describes improvements in solution polishing

Locale and the writing of readable files

Describes impact of locale on the writing of files.

Conversion overview

For users of ILOG CPLEX 11.0, the following topics offer guidelines for easy migration to ILOG CPLEX 11.1.

Users of prior versions must first apply the Conversion Notes accompanying previous versions of ILOG CPLEX before upgrading to this one. For easy reference, the ILOG CPLEX 11.0 Release notes are repeated in this publication, following these release notes for ILOG CPLEX 11.1.

The following topics cover changes that may affect all users of ILOG CPLEX 11.1.

Solution polishing and MIP starts

ILOG CPLEX 11.1 offers improvements in solution polishing. In most cases, solution polishing can now improve a MIP start (that is, a solution suitable as an advanced starting point) even if that MIP start is not consistent with dual presolve reductions that ILOG CPLEX makes.

Models and applications that formerly required special settings of the preprocessing parameters to turn off dual presolve reductions, nonlinear presolve reductions, or symmetry reductions may no longer require those special settings.

In the rare event that solution polishing is not able to improve a MIP start that you provide, consider disabling one or more of these parameters as documented in the *ILOG CPLEX Parameter Reference Manual*:

- ◆ the *presolve switch*: PreInd, CPX_PARAM_PREIND
- ◆ the *linear reduction switch*: PreLinear, CPX_PARAM_PRELINEAR
- ◆ the *symmetry breaking* parameter: Symmetry, CPX_PARAM_SYMMETRY

Solution polishing in the *ILOG CPLEX User's Manual* now includes more information about how to use solution polishing effectively. In particular, this revised topic gives more information about the relationship between branch & cut and solution polishing. It also specifies the stopping criteria for solution polishing, and introduces the other parameters governing solution polishing. Additionally, it provides detailed information about how to use solution polishing in several common situations.

Locale and the writing of readable files

ILOG CPLEX now ignores the setting of the locale when it writes a file in a format that it can read later. (For a description of file formats that ILOG CPLEX reads and writes, see the *ILOG CPLEX File Formats Reference Manual*.)

Previously, when ILOG CPLEX took the locale into account when it wrote files that it might read again later, the resulting file could contain inconsistent representations of data, rendering the written file unreadable by ILOG CPLEX later. In general, ILOG CPLEX cannot read files written with a locale where conventions differ from the standard representations in the C programming language.

ILOG CPLEX continues to write log files according to the locale set by Callable Library or Concert Technology applications.

Conversion notes for Concert Technology users

A new version of Concert Technology (Concert Technology 2.6) accompanies this release of ILOG CPLEX 11.1. This new version requires you to **recompile** and **link** your Concert Technology applications.

Conversion notes for Callable Library users

Just **recompile** and **link** your application with ILOG CPLEX 11.1 Callable Library. In addition, this planned change in the Callable Library may affect existing ILOG CPLEX applications in the future.

Deprecated C routines

The following routines of the Callable Library (C API) have been undocumented since CPLEX 5.0. They are now **deprecated**. They are no longer recommended for use in your applications. In a future release, they will be removed from the product.

- ◆ CPXlpread
- ◆ CPXlpmread
- ◆ CPXlpqread
- ◆ CPXmpsread
- ◆ CPXmpsmread
- ◆ CPXmpsquad
- ◆ CPXsavread
- ◆ CPXsavmread
- ◆ CPXsavqread
- ◆ CPXmbaseread

ILOG recommends that you use `CPXreadcopyprob` instead, followed by a query of the `CPXLP` object for your target data by means of such query routines as `CPXgetrows`, `CPXgetobjsense`, `CPXgetsense`. Examples demonstrating the use of `CPXreadcopyprob` are available in the samples distributed with this version of the product in *yourCPLEXhome/examples*.

ILOG CPLEX 11.0

Dedicated to Lloyd Clarke (1964–2007)

In this section

ILOG CPLEX 11.0 Release Notes

These release notes highlight improvements and new features in ILOG CPLEX 11.0.

Announcements: port changes

Highlights news from the Marketing Manager.

Conversion notes for all users

Describes changes important to all customers.

Conversion notes for Concert Technology users

Describes changes of interest to customers who use Concert Technology.

Conversion notes for Callable Library users

Describes changes of interest to customers who use the Callable Library (C API).

New features for all users

Documents new features in ILOG CPLEX 11.0.

New features in Concert Technology

Describes new features of Concert Technology.

New features in the Callable Library

Describes new features of the Callable Library (C API).

New features in the Interactive Optimizer

Describes new features of the Interactive Optimizer.

Documentation for IDEs

Announces additional support for ILOG CPLEX documentation in integrated development environments.

ILOG CPLEX 11.0 Release Notes

Please review these notes before using ILOG CPLEX 11.0.

Announcements: port changes

For a complete list of machine types and library formats (including version numbers of compilers and JDKs) see the file *yourCPLExhome* /mptable.html .

Mac OS Darwin port available

ILOG CPLEX 11.0 is available on an additional x86 platform: Mac OS X 10.4, also known as Darwin 8, with the GNU g++ 4.0 (32-bit) compiler and JDK 5.0.

JDK 1.4 deprecated

JDK 1.4, also known as J2EE 1.4, has been **deprecated**. That is, this development kit will no longer be supported after ILOG CPLEX 11.0. Instead, ILOG urges customers who currently use this deprecated development kit to migrate to a more recent version, such as JDK 6, as recommended by Sun Microsystems.

Visual Studio .NET 2003 deprecated

Visual Studio .NET 2003 has been **deprecated**. That is, this compiler version will no longer be supported after ILOG CPLEX 11.0. Instead, ILOG urges customers who currently use this deprecated compiler to migrate to a more recent version, such as Visual Studio 8.

Conversion notes for all users

Describes changes important to all customers.

In this section

Overview

Introduces migration from earlier versions.

Changes in existing parameters

Describes changes in existing parameters of interest to all customers.

Dynamic search

Describes a major improvement in search facilities.

Informational callbacks

Describes a new kind of callback, appropriate for dynamic search.

Node limits

Describes a change in behavior for the node limit parameter.

File format removed: SOS

Announces removal of a deprecated file format: SOS.

Error codes removed

Lists error codes that are no longer relevant.

Error codes added

Lists error codes new in this release.

Overview

For users of earlier versions of ILOG CPLEX, the following topics offer guidelines for easy migration to ILOG CPLEX 11.0. (Users of prior versions must first apply the Conversion Notes accompanying previous versions of ILOG CPLEX before upgrading to this one.)

The following topics cover changes that may affect all users of ILOG CPLEX 11.0.

Changes in existing parameters

The table **Changed parameters** lists parameters that have changed in behavior or possible values since the previous release. See the documentation of the changed parameter in the *ILOG CPLEX Parameters Reference Manual* for more detail about each parameter.

Changed parameters

Reference Manual	Concert Parameter	Callable Library	Interactive Optimizer	Comment
<i>advanced start switch</i>	AdvInd	CPX_PARAM_ADVIND	advance	setting 2 has new effect for MIP
<i>integrality tolerance</i>	EpInt	CPX_PARAM_EPINT	mip tolerances integrality	maximum value is now 0.5
<i>symmetry breaking</i>	Symmetry	CPX_PARAM_SYMMETRY	preprocessing symmetry	new more aggressive settings: 4, 5
<i>MIP node limit</i>	NodeLim	CPX_PARAM_NODELIM	mip limits nodes	behavior has changed for 0 (zero) and 1 (one)
<i>global default thread count</i>	Threads	CPX_PARAM_THREADS	threads	new default: instead of 1, the default depends on the number of threads licensed and the number available on the computer; interaction with new parallel parameter
REMOVED	MIPThreads	CPX_PARAM_MIPTHEADS	mip limits threads	REMOVED: use threads parameter
REMOVED	BarThreads	CPX_PARAM_BARIHEADS	barrier limits threads	REMOVED: use threads parameter
REMOVED	StrongThreadLim	CPX_PARAM_STRONGTHREALIM	mip limits strongthreads	REMOVED: use threads parameter

Dynamic search

ILOG CPLEX 11.0 offers a new and innovative algorithmic approach for mixed integer programs. This new feature, known as dynamic search, is available by default. For many models, dynamic search finds feasible and optimal solutions more quickly than conventional branch & cut.

A new parameter (`MIPSearch`, `CPX_PARAM_MIPSEARCH`) allows you to turn off this feature in applications where you judge that conventional branch & cut search is more appropriate.

Query callbacks and control callbacks are not compatible with dynamic search. (For more about this point, see *Informational callbacks* in these release notes.) Consequently, if a callback attempts to alter the search path while dynamic search is in progress, CPLEX will raise the error `CPXERR_MIPSEARCH_WITH_CALLBACKS`. For more about dynamic search and callbacks, see also *Using optimization callbacks* in the *ILOG CPLEX User's Manual*.

Informational callbacks

Informational callbacks, new in ILOG CPLEX 11.0, provide a simplified approach to monitoring progress of your MIP solve. Informational callbacks are compatible with recent ILOG innovations such as dynamic search and deterministic parallel MIP optimization.

In previous versions of ILOG CPLEX, only query and control callbacks were available. Query callbacks (also known as diagnostic callbacks) and control callbacks are not compatible with dynamic search (the new default search). At default settings, in the presence of query callbacks or control callbacks, ILOG CPLEX 11.0 turns off dynamic search and turns off deterministic parallel MIP optimization.

In Concert Technology, new informational callback classes are available. For samples of their use, see:

- ◆ `ilomipex4.cpp` in the C++ API
- ◆ `MIPex4.java` in the Java API
- ◆ `MIPex4.cs` or `MIPex4.vb` in the .NET API

In the Callable Library (C API), new routines, `CPXsetinfocallbackfunc` and `CPXgetinfocallbackfunc`, enable you to install and access informational callbacks. For a sample of their use, see `mipex4.c`.

The error `CPXERR_IN_INFOCALLBACK` will be raised if a callback calls any routine other than `CPXgetcallbackincumbent` or `CPXgetcallbackinfo` (the two routines allowed in informational callbacks in the C API).

Examples illustrating the use of callbacks have been revised to show you both how to replace old style callbacks with informational callbacks and how to set search parameters in the presence of callbacks. These revised examples are available in the standard distribution in `yourCPLEXhome/examples` for all APIs.

Using optimization callbacks in the *ILOG CPLEX User's Manual* has been extensively revised to document these changes in callbacks and their interaction with dynamic search. Likewise, *Parallel optimizers* in the *ILOG CPLEX User's Manual* has been revised with respect to callbacks and parallel optimization.

Node limits

Behavior of the node limit parameter (NodeLim, CPX_PARAM_NODELIM) has changed.

Now when this parameter is set to 0 (zero), ILOG CPLEX completes processing at the root; that is, cuts are created and heuristics are applied at the root, but no nodes are created. When the parameter is set to 1 (one), ILOG CPLEX branches at the root; that is, nodes are created but not solved.

Previously, 0 (zero) solved the initial root LP relaxation and stopped before applying any cuts, heuristics, or other processing at the root node. The behavior for all other settings (including the value of the default) remains unchanged.

File format removed: SOS

The SOS file format, deprecated in a previous release, has been removed. Instead of this formatted, ASCII-based file format for special ordered sets (SOS), ILOG recommends MPS or LP format. The *ILOG CPLEX File Formats Reference Manual* documents extensions of these formats to support special ordered sets (SOS).

The following routines of the Callable Library are no longer available:

- ◆ CPXsoswrite
- ◆ CPXsosread
- ◆ CPXreadcopysos

Error codes removed

The following error codes have been removed. Conditions that gave rise to these codes no longer exist.

- ◆ 1269 CPXERR_ABORT_CONDITION_NO
- ◆ 1100 CPXERR_BOUNDS_INFEAS
- ◆ 3011 CPXERR_BOUNDS_INT
- ◆ 3022 CPXERR_SEMI_BDS

Error codes added

The following error codes have been added.

- ◆ 1123 CPXERR_PRESLV_TIME_LIM
- ◆ 1234 CPXERR_THREAD_FAILED
- ◆ 1804 CPXERR_IN_INFOCALLBACK
- ◆ 1805 CPXERR_MIPSEARCH_WITH_CALLBACKS
- ◆ 1806 CPXERR_LP_NOT_IN_ENVIRONMENT
- ◆ 3024 CPXERR_NO_SOLNPOOL
- ◆ 3414 CPXERR_FILTER_VARIABLE_TYPE

Conversion notes for Concert Technology users

Describes changes of interest to customers who use Concert Technology.

In this section

Overview

Introduces changes of interest to users of Concert Technology.

Conversion notes for users of the C++ API

Describes changes in the C++ API.

Conversion notes for users of the Java API

Describes changes in the Java API.

Conversion notes for users of the .NET API

Describes changes in the .NET API.

Overview

A new version of Concert Technology (Concert Technology 2.5) accompanies this release of ILOG CPLEX 11.0. This new version requires you to recompile and relink your Concert Technology applications. In addition, the following conversion notes are of interest to users of Concert Technology.

Conversion notes for users of the C++ API

The method `IloSolution::isBound` of the C++ API has been **deprecated**. It will be removed in a future release. If you are using this deprecated method, ILOG recommends that you use `IloSolution::isFixed` instead.

The following methods of the C++ API were deprecated in ILOG CPLEX 8.0 and ILOG CPLEX 10.1. They have been **removed** from ILOG CPLEX 11.0. Use the parameters `RootAlgorithm` and `NodeAlgorithm` instead of these deprecated methods.

- ◆ `IloCplex::getRootAlgorithm`
- ◆ `IloCplex::setRootAlgorithm`
- ◆ `IloCplex::getNodeAlgorithm`
- ◆ `IloCplex::setNodeAlgorithm`

The following methods of the C++ API were deprecated in a previous release of ILOG Concert Technology. They have been **removed** from ILOG Concert Technology 2.5 C++ API.

- ◆ `IloRange::setCoef(const IloNumVar var, IloNum val)`
- ◆ `IloRange::setCoef(const IloNumVarArray vars, IloNumArray coefs)`
- ◆ `IloObjective::setCoef(const IloNumVar var, IloNum value)`
- ◆ `IloObjective::setCoef(const IloNumVarArray vars, IloNumArray vals)`

The following methods were deprecated in a previous release; they have been **removed**; use `IloCplex::getNcuts` instead with the enum `IloCplex::CutType`.

- ◆ `IloCplex::getNcovers`
- ◆ `IloCplex::getNcliques`

Conversion notes for users of the Java API

The following methods of the Java API were deprecated in ILOG CPLEX 8.0 and ILOG CPLEX 10.1. They have been **removed** from ILOG CPLEX 11.0. Use the parameters `RootAlgorithm` and `NodeAlgorithm` instead of these deprecated methods.

- ◆ `IloCplex.getRootAlgorithm`
- ◆ `IloCplex.setRootAlgorithm`
- ◆ `IloCplex.getNodeAlgorithm`
- ◆ `IloCplex.setNodeAlgorithm`

These methods for counting cuts were deprecated in a previous release of ILOG Concert Technology; they have been removed from the Java API; use `IloCplex.getNcuts` instead with the enum `IloCplex.CutType`.

- ◆ `IloCplex.getNcliques`
- ◆ `IloCplex.getNcovers`

Conversion notes for users of the .NET API

These members for counting cuts were deprecated in a previous release of ILOG Concert Technology; they have been removed from the .NET API; use the member `Cplex.Ncuts` instead with the enum `Cplex.CutType`.

◆ `Cplex.Ncliques`

◆ `Cplex.Ncovers`

Conversion notes for Callable Library users

These changes in the Callable Library may affect existing ILOG CPLEX applications.

Environment and problem mismatch raises an error

In a Callable Library (C API) application, any attempt to use a problem object in an environment other than the environment where the problem object was created will raise the error `CPXERR_LP_NOT_IN_ENVIRONMENT`.

In the Callable Library, the `lp` pointer to a problem object (regardless of the type of the problem) returned by `CPXcreateprob` must remain within the environment returned by `CPXopenCPLEX`.

CPXnewcols default upper bound

`CPXnewcols (env, lp, 1, &obj, NULL, NULL, "B", "newbin")` now creates a new binary variable `newbin` with objective `obj`, lower bound 0 (zero) and upper bound 1 (one). Formerly, the upper bound was infinite by default.

Replacement for removed cut-counting routines

These routines, deprecated in a previous release of ILOG CPLEX, have been removed from the Callable Library (C API):

- ◆ `CPXgetclqcnt`
- ◆ `CPXgetgenclqcnt`
- ◆ `CPXgetcovcnt`

Use `CPXgetnumcuts` and the symbolic constants to specify the type of cut instead.

- ◆ `CPX_CUT_COVER`
- ◆ `CPX_CUT_GUBCOVER`
- ◆ `CPX_CUT_FLOWCOVER`
- ◆ `CPX_CUT_CLIQUE`
- ◆ `CPX_CUT_FRAC`
- ◆ `CPX_CUT_MIR`
- ◆ `CPX_CUT_FLOWPATH`
- ◆ `CPX_CUT_DISJ`

- ◆ CPX_CUT_IMPLBD
- ◆ CPX_CUT_ZEROHALF
- ◆ CPX_CUT_LOCALCOVER
- ◆ CPX_CUT_TIGHTEN
- ◆ CPX_CUT_OBJDISJ
- ◆ CPX_CUT_USER
- ◆ CPX_CUT_TABLE
- ◆ CPX_CUT_SOLNPOOL

New features for all users

Documents new features in ILOG CPLEX 11.0.

In this section

Overview

Introduces major new features of this release.

Solution pool

Announces the solution pool, a new feature for accumulating multiple solutions for a MIP.

Tuning tool

Announces the tuning tool, a feature to analyze performance with respect to parameter settings.

Deterministic parallel MIP

Announces the new deterministic parallel MIP optimizer.

New algorithm for MIQCPs

Announces a new algorithm for MIQCPs.

Feasibility pump

Announces an implementation of the feasibility pump.

New status code for FeasOpt

Announces a new status code appropriate for FeasOpt.

New status codes for solution pool

Announces a new status code appropriate for MIP optimizer or populate, the solution pool optimizer.

Solution information available for MIPs and FeasOpt

Describes an extension to access solution information after MIP optimization or FeasOpt.

Interruption and termination

Describes a change in behavior with respect to interruption and termination.

New parameters

Lists new parameters with links to further documentation.

Overview

In addition to the new features introduced with their implications for conversion in *Dynamic search* and *Informational callbacks*, here is more detail about other major new features in this release.

Solution pool

A new feature of ILOG CPLEX 11.0, known as the solution pool, enables you to generate and store multiple solutions to a mixed integer programming (MIP) problem. This new feature is available in all application programming interfaces (APIs) as well as in the Interactive Optimizer.

For an introduction to the solution pool, see the new topic *Solution pool: generating and keeping multiple solutions* of the *ILOG CPLEX User's Manual*. That introduction shows you how to use new methods or routines to generate multiple solutions and keep them in the solution pool. It also explains the interaction between conventional MIP optimization and the solution pool.

In addition, that new topic of the user's manual also demonstrates how to use new parameters to manage the solution pool, as documented in the *ILOG CPLEX Parameters Reference Manual*:

- ◆ *limit on number of solutions generated for solution pool*
- ◆ *limit on number of solutions kept in solution pool*
- ◆ *solution pool intensity*
- ◆ *relative gap for solution pool*
- ◆ *absolute gap for solution pool*
- ◆ *solution pool replacement strategy*

Sample applications in `yourCPLEXhome/examples` show you how to invoke the populate procedure and how to manage the solution pool.

- ◆ `ilopopulate.cpp` in the C++ API
- ◆ `Populate.java` in the Java API
- ◆ `Populate.cs` or `Populate.vb` in the .NET API
- ◆ `populate.c` in the Callable Library (C API)

There are also examples in that new topic of the user's manual to show various approaches to filtering the solutions accumulated in the solution pool. For additional information about solution pool filters, see also *Filtering the solution pool* in the *ILOG CPLEX User's Manual* and *FLT file format: filter files for the solution pool* in the *ILOG CPLEX File Format Reference Manual*.

Tuning tool

To help you analyze whether default parameter settings provide the best performance for your particular model or models and to aid you in adjusting parameter settings efficiently, ILOG CPLEX 11.0 offers a new performance tuning tool. This tool is available in all application programming interfaces (APIs) as well as in the Interactive Optimizer.

For an introduction to the tuning tool, see the new topic *Tuning tool* of the *ILOG CPLEX User's Manual*. That chapter includes sample sessions in the Interactive Optimizer, as well as references to examples available in the standard distribution of the product.

Additionally, there are sample applications using the tuning tool in `yourCplexhome/examples`:

- ◆ `ilotuneset.cpp` in the C++ API
- ◆ `TuneSet.java` in the Java API
- ◆ `TuneSet.cs` or `TuneSet.vb` in the .NET API
- ◆ `tuneset.c` in the Callable Library (C API)

Deterministic parallel MIP

ILOG CPLEX 11.0 extends the capabilities of the parallel MIP optimizer to include a new deterministic mode of operation in addition to the existing opportunistic mode.

In this context, **deterministic** means that multiple runs with the same model at the same parameter settings on the same platform will reproduce the same solution path and results. In contrast, opportunistic search skips synchronization between threads and thus entails less waiting and consequently performs faster. Lack of synchronization often provides improved performance but leads to variations in the search path and consequently possibly different solution vectors, though not a difference in the optimal objective value.

A new parameter (`ParallelMode` or `CPX_PARAM_PARALLELMODE`) allows you to control which mode ILOG CPLEX uses to solve your model. This new parameter has three possible settings: automatic (where ILOG CPLEX decides which mode to apply), deterministic, and opportunistic. In most cases, the automatic setting of this parameter selects deterministic mode if your platform supports multiple threads. For more detail about this parameter, its settings, and their interactions with other parameters, see *parallel mode switch* in the *ILOG CPLEX Parameters Reference Manual*.

The topic *Parallel optimizers* in the *ILOG CPLEX User's Manual* has been extensively revised to document changes due to the introduction of deterministic parallel mode. See particularly *Parallel MIP optimizer* for more detail.

New algorithm for MIQCPs

ILOG CPLEX 11.0 offers a new algorithm for the solution of quadratically constrained mixed integer programs (MIQCPs). A new parameter (MIQCPStrat, CPX_PARAM_MIQCPSTRAT) sets the strategy that ILOG CPLEX uses to solve these problems.

This parameter lets you control the type of relaxation used to solve a MIQCP. At the default setting of 0 (zero), ILOG CPLEX automatically chooses a strategy based on characteristics of your problem.

When you set this parameter to the value 1 (one), you tell ILOG CPLEX to solve a **QCP relaxation** of the model at each node. This is the behavior available in previous versions of ILOG CPLEX.

When you set this parameter to the value 2, you tell ILOG CPLEX to attempt to solve only an **LP relaxation** of the model at each node.

For documentation of this parameter, see *MIQCP strategy switch* in the *ILOG CPLEX Parameters Reference Manual*.

For more general information about the types of quadratically constrained models that ILOG CPLEX solves, see *Identifying a quadratically constrained program (QCP)* in the *ILOG CPLEX User's Manual*.

Feasibility pump

ILOG CPLEX 11.0 makes the feasibility pump heuristic available to you through a new parameter (FPHeur, CPX_PARAM_FPHEUR). This heuristic has demonstrated its ability to find an initial feasible solution even in certain very hard mixed integer programming problems (MIPs). This new parameter allows you to turn on or off the feasibility pump; the parameter also allows you to specify whether the feasibility pump concentrates on feasibility or improved objective values as it searches for a feasible solution.

For documentation of this new parameter, see the topic titled *feasibility pump switch* in the *ILOG CPLEX Parameters Reference Manual*.

For more detail about the feasibility pump heuristic, see research reported by Fischetti, Glover, and Lodi (2003, 2005), by Bertacco, Fischetti, and Lodi (2005), and by Achterberg and Berthold (2005, 2007).

New status code for FeasOpt

In each application programming interface (API), there is a new status code. This new status code is returned whenever FeasOpt in phase one finds a model to be feasible and installs a feasible solution.

- ◆ In the C++ API, the enumeration `IloCplex::CplexStatus` includes the new status `Feasible`.
- ◆ In the Java API, the enumeration `IloCplex.CplexStatus` includes the new status `Feasible`.
- ◆ In the .NET API, the enumeration `Cplex.CplexStatus` includes the new status `Feasible`.
- ◆ In the Callable Library, the macros `CPX_STAT_FEASIBLE` for continuous (LP) models and `CPXMIP_FEASIBLE` for discrete (MIP) models report the new status.

New status codes for solution pool

In each application programming interface (API), there are new status codes appropriate for the solutions in the solution pool. These new status codes are returned after MIP optimization or populate, the solution pool optimizer.

- ◆ In the C++ API, the enumeration `IloCplex::CplexStatus` includes the new status codes `PopulateSolLim`, `OptimalPopulated`, `OptimalPopulatedTol`.
- ◆ In the Java API, the enumeration `IloCplex.CplexStatus` includes the new status codes `PopulateSolLim`, `OptimalPopulated`, `OptimalPopulatedTol`.
- ◆ In the .NET API, the enumeration `Cplex.CplexStatus` includes the new status codes `PopulateSolLim`, `OptimalPopulated`, `OptimalPopulatedTol`.
- ◆ In the Callable Library, the macros `CPXMIP_POPULATESOL_LIM`, `CPXMIP_OPTIMAL_POPULATED`, `CPXMIP_OPTIMAL_POPULATED_TOL` represent these new status codes.

Solution information available for MIPs and FeasOpt

Existing methods in Concert Technology, such as `isPrimalFeasible` or `isDualFeasible`, and existing routines to access solution information about continuous models in the Callable Library (C API), such as `CPXsolninfo`, have been extended so that solution information is available in ILOG CPLEX 11.0 for mixed integer programming problems (MIPs) as well as for FeasOpt in all APIs. See the documentation of these methods and routines in the reference manuals of the APIs for detail.

Interruption and termination

ILOG CPLEX 11.0 has been improved, so that a user's application can intercept interrupt-signals and direct CPLEX to behave appropriately.

In **Concert Technology**, there is a new class of objects representing termination. These objects to terminate optimization can be attached to a CPLEX object and then called from inside or outside callbacks generally. These objects are useful in place of control callbacks, for example, when the user wants to terminate the search rather than redirect it.

A new class supports this behavior in Concert Technology:

- ◆ `IloCplex::Aborter` in the C++ API
- ◆ `IloCplex.Aborter` in the Java API
- ◆ `Cplex.Aborter` in the .NET API

In the **Callable Library**, the new routine `CPXsetterminate` is available to enable your application to interrupt, abort, or terminate ILOG CPLEX gracefully.

New parameters

The release notes introduce these new parameters. More detail about each of them is available in the *ILOG CPLEX Parameters Reference Manual* and in the *ILOG CPLEX User's Manual*.

New parameters

Parameter in Concert Technology	Parameter in Callable Library	Parameter in Interactive Optimizer	Reference Manual
EachCutLim	CPX_PARAM_EACHCUTLIM	mip limit eachcutlimit	<i>type of cut limit</i>
FPHeur	CPX_PARAM_FPHEUR	mip strategy fpheur	<i>feasibility pump switch</i>
MIPSearch	CPX_PARAM_MIPSEARCH	mip strategy search	<i>MIP dynamic search switch</i>
MIQCPStrat	CPX_PARAM_MIQCPSTRAT	mip strategy miqcpstrat	<i>MIQCP strategy switch</i>
ParallelMode	CPX_PARAM_PARALLELMODE	parallel	<i>parallel mode switch</i>
PopulateLim	CPX_PARAM_POPULATELIM	mip limits populate	<i>limit on number of solutions generated for solution pool</i>
SolnPoolCapacity	CPX_PARAM_SOLNPOOLCAPACITY	mip pool capacity	<i>limit on number of solutions kept in solution pool</i>
SolnPoolIntensity	CPX_PARAM_SOLNPOOLINTENSITY	mip pool intensity	<i>solution pool intensity</i>
SolnPoolReplace	CPX_PARAM_SOLNPOOLREPLACE	mip pool replace	<i>solution pool replacement strategy</i>
SolnPoolGap	CPX_PARAM_SOLNPOOLGAP	mip pool relgap	<i>relative gap for solution pool</i>
SolnPoolAGap	CPX_PARAM_SOLNPOOLAGAP	mip pool absgap	<i>absolute gap for solution pool</i>
TuningDisplay	CPX_PARAM_TUNINGDISPLAY	tune display	<i>tuning information display</i>
TuningMeasure	CPX_PARAM_TUNINGMEASURE	tune measure	<i>tuning measure</i>
TuningRepeat	CPX_PARAM_TUNINGREPEAT	tune repeat	<i>tuning repeater</i>
TuningTiLim	CPX_PARAM_TUNINGTILIM	tune timelimit	<i>tuning time limit</i>
ZeroHalfCuts	CPX_PARAM_ZEROHALFCUTS	mip cuts zerohalfcut	<i>MIP zero-half cuts switch</i>

New features in Concert Technology

Describes new features of Concert Technology.

In this section

Overview

Highlights new features specific to Concert Technology.

Overview

In addition to the new features of general interest to all users of ILOG CPLEX 11.0, there are new features in Concert Technology:

- ◆ A method to **count cuts** has been introduced to replace those methods deprecated and removed in previous versions. This method offers a uniform approach to counting all types of cuts by allowing you to specify the type(s) of interest to you.
 - `IloCplex::getNcuts` in the C++ API
 - `IloCplex.getNcuts` in the Java API
 - `Cplex.GetNcuts` in the .NET API
- ◆ There is a new **output operator** for the enumeration `IloCplex::CplexStatus` in the C++ API.
- ◆ There is a new class of **visitors** for extractable objects `IloExtractableVisitor` in the C++ API. Instances of this class support introspection of extractable objects in an instance of `IloModel`.
- ◆ A group of **parameters** can be treated as a **set**. To aid you in managing the parameters in a set, ILOG CPLEX provides an **iterator** over a parameter set.
 - In the C++ API, the iterator is an instance of the class `IloCplex::ParameterSet::Iterator`.
 - In the Java API, the iterator is an instance of the class `java.util.Iterator`.
 - In the .NET API, the iterator is an instance of the class `System.Collections.IEnumerator`.

For samples of a parameter set and its iterator in use, see these examples in the standard distribution of the product:

- `ilotuneset.cpp` in the C++ API
- `TuneSet.java` in the Java API
- `TuneSet.cs` or `TuneSet.vb` in the .NET API

New features in the Callable Library

In addition to the new features of general interest to all users of ILOG CPLEX migration, the following topics may be of particular interest to users of the Callable Library (C API).

Counting cuts

The new routine `CPXgetnumcuts` replaces the deprecated and removed routines to count the number of cuts. This new routine offers a uniform interface allowing you to specify which types of cuts are of interest to you. It thus eliminates the need for multiple routines, one to count each type of cut.

Data cleaning tool in the Callable Library

ILOG CPLEX 11.0 offers a tool to change small values of coefficients in a model to zero. You may find this tool helpful in reducing numerical error introduced by floating-point and finite-precision arithmetic. The practice implemented by this tool is also known as **zero-ing out negligible coefficients**. In the Callable Library (C API), use the routine `CPXcleanup`. It accepts a tolerance (that is, an epsilon value) within which small coefficients will be zero-ed out in a problem that your application has already created with `CPXcreateprob`.

Managing parameters in the Callable Library

For users of the Callable Library, there are two new routines to manage parameters:

- ◆ `CPXgetchgparam` to access a list of parameters not at their default value;
- ◆ `CPXgetparamtype` to access the type of a specified parameter.

These routines may be useful with the tuning tool. Both routines appear in the example `tuneset.c`.

New features in the Interactive Optimizer

In addition to the new features of general interest to all users of ILOG CPLEX migration, the following topics may be of particular interest to users of the Interactive Optimizer:

Data cleaning tool in the Interactive Optimizer

ILOG CPLEX 11.0 offers a tool to change small values of coefficients in a model to zero. You may find this tool helpful in reducing numerical error introduced by floating-point and finite-precision arithmetic. The practice implemented by this tool is also known as **zero-ing out negligible coefficients**.

In the **Interactive Optimizer**, after you have read or entered a problem and detected negligible values among the coefficients, use the command `change values`. The Interactive Optimizer will then prompt you for a tolerance (that is, an epsilon value) within which you want small values to be changed to 0 (zero).

Zero-ing out the objective

It is now possible to set the value of coefficients in the objective function to zero in the Interactive Optimizer. (This change was already available in the APIs of ILOG CPLEX.) This practice is also known familiarly as **zero-ing out the objective**. The command to use in the Interactive Optimizer to achieve this effect is the following:

```
change delete constraint 0
```

In a problem in which the objective function includes a quadratic term and thus is a problem of type QP, this change of setting the objective function to zero will result in a change as well in the **type of problem**.

Documentation for IDEs

ILOG CPLEX 11.0 provides reference documentation that can be integrated into and accessed from integrated development environments (IDEs) such as Microsoft Visual Studio or Eclipse.

For users of **Microsoft Visual Studio**, the documentation to support IntelliSense is available (as in previous releases) in the XML files accompanying `ILOG.CPLEX.dll` and `ILOG.Concert.dll`.

For users of **Eclipse**, the documentation to support tooltips is available:

- ◆ on UNIX and GNU/Linux platforms, in `yourCPLEXinstallation/doc/doc/html/refjavacplex/html`
- ◆ on Microsoft Windows, in `C:\ILOG\CPLEX110\doc\refjavacplex`

ILOG CPLEX Getting Started

Introduces ILOG CPLEX through tutorials and examples.

ILOG CPLEX Getting Started

Introduces ILOG CPLEX through tutorials and examples.

In this section

Introducing ILOG CPLEX

This preface introduces ILOG CPLEX.

Setting up ILOG CPLEX

Shows how to set up ILOG CPLEX and how to check the installation. It includes information for users of all platforms.

Tutorials

This part provides tutorials to introduce you to each of the components of ILOG CPLEX.

Introducing ILOG CPLEX

This preface introduces ILOG CPLEX.

In this section

What is ILOG CPLEX?

Describes ILOG CPLEX, its components, and options.

Using the parallel optimizers

Introduces the parallel optimizers available in ILOG CPLEX.

Data entry options

Introduces the means of data entry that ILOG CPLEX supports.

What ILOG CPLEX is not

Contrasts ILOG CPLEX with other tools.

What you need to know

Suggests prerequisites for effective use of ILOG CPLEX.

What's in this manual

Outlines the information available in this manual.

Notation in this manual

Describes the conventions of notation in this manual.

Related documentation

Describes additional documentation available for ILOG CPLEX.

What is ILOG CPLEX?

Describes ILOG CPLEX, its components, and options.

In this section

Overview

Defines the kind of problems that ILOG CPLEX solves.

ILOG CPLEX components

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

Optimizer options

Introduces the options available in ILOG CPLEX.

Overview

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

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

where \sim can be $<$, $>$, or $=$, and the upper bounds u_i and lower bounds l_i may be positive infinity, negative infinity, or any real number.

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

$$\begin{array}{ll} \text{Objective function coefficients} & c_1, c_2, \dots, c_n \\ \\ \text{Constraint coefficients} & a_{11}, a_{21}, \dots, a_{n1} \\ & \dots \\ & a_{m1}, a_{m2}, \dots, a_{mn} \\ \\ \text{Righthand sides} & b_1, b_2, \dots, b_m \\ \\ \text{Upper and lower bounds} & u_1, u_2, \dots, u_n \text{ and } l_1, l_2, \dots, l_n \end{array}$$

The optimal solution that ILOG CPLEX computes and returns is:

$$\text{Variables } x_1, x_2, \dots, x_n$$

ILOG CPLEX also can solve several extensions to LP:

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

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

ILOG CPLEX components

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

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

Concert Technology libraries

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

The ILOG Concert Technology libraries make use of the Callable Library (described next).

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

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

Compatible platforms

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

Installation

If you have not yet installed ILOG CPLEX on your platform, consult *Setting up ILOG CPLEX*. It contains instructions for installing ILOG CPLEX.

Optimizer options

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

Default settings will result in a call to an optimizer that is appropriate to the class of problem you are solving. However you may wish to choose a different optimizer for special purposes. An LP or QP problem can be solved using any of the following CPLEX optimizers: Dual Simplex, Primal Simplex, Barrier, and perhaps also the Network Optimizer (if the problem contains an extractable network substructure). Pure network models are all solved by the Network Optimizer. QCP models, including the special case of SOCP models, are all solved by the Barrier optimizer. MIP models are all solved by the Mixed Integer Optimizer, which in turn may invoke any of the LP or QP optimizers in the course of its computation. *Optimizers* summarizes these possible choices.

Optimizers

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

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

Using the parallel optimizers

Parallel Barrier, Parallel MIP, and Concurrent optimizers are implemented to run on hardware platforms with parallel processors. These parallel optimizers can be called from the Interactive Optimizer and the Component Libraries, if your application is licensed for parallel optimization.

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

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

Data entry options

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

Data entry options are described briefly in this manual. File formats are documented in the reference manual *ILOG CPLEX File Formats Reference Manual*.

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

What ILOG CPLEX is not

ILOG CPLEX is not a modeling language, nor is it an integrated development environment (IDE). You can completely model and solve your optimization problems with ILOG CPLEX; however, the features it provides do not offer the interactive facilities of a modeling system in an integrated development environment. If you are looking for such a system, consider ILOG OPL and ILOG ODM.

ILOG OPL Development Studio is a powerful system for rapid development and deployment of optimization applications. ILOG OPL Development Studio (or OPL for short) consists of:

- ◆ the Optimization Programming Language (OPL) for developing optimization models;
- ◆ an Integrated Development Environment (IDE) to execute and test optimization models;
- ◆ a command line tool (oplrn) to execute models from the command line;
- ◆ Application Programming Interfaces (APIs) to embed models into standalone applications;
- ◆ support for ILOG Optimization Decision Manager (ODM).

ILOG OPL Development Studio is integrated with a companion product called ILOG Optimization Decision Manager (ODM). ODM is both a tool for application development and a runtime environment. The combined product is a complete solution for the development and deployment of optimization-based planning and scheduling applications. Applications built with ODM allow users to adjust assumptions, operating constraints, and goals for planning and scheduling resources. End users of ODM also see the results in familiar business terminology. ODM applications support extensive what-if analysis and scenario comparison features "out of the box."

What you need to know

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

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

What's in this manual

Setting up ILOG CPLEX tells how to install CPLEX.

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

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

Concert Technology tutorial for C++ users, describes the same activities using the classes in the C++ implementation of the CPLEX Concert Technology Library.

Concert Technology tutorial for Java users, describes the same activities using the classes in the Java implementation of the CPLEX Concert Technology Library.

Concert Technology tutorial for .NET users, describes the same activities using .NET facilities.

Callable Library tutorial, describes the same activities using the routines in the ILOG CPLEX Callable Library.

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

Notation in this manual

This manual observes the following conventions in notation and names.

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

Related documentation

In addition to this introductory manual, the standard distribution of ILOG CPLEX comes with the *ILOG CPLEX User's Manual* and the *ILOG CPLEX Reference Manuals*. All ILOG documentation is available online in hypertext mark-up language (HTML). It is delivered with the standard distribution of the product and accessible through conventional HTML browsers for customers on most platforms. For customers on Microsoft Windows, the documentation is available as Microsoft Compiled HTML (also known as CHM).

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

As you work with ILOG CPLEX on a long-term basis, you should read the complete *User's Manual* to learn how to design models and implement solutions to your own problems. Consult

the reference manuals for authoritative documentation of the Component Libraries, their application programming interfaces (APIs), and the Interactive Optimizer.

Setting up ILOG CPLEX

Shows how to set up ILOG CPLEX and how to check the installation. It includes information for users of all platforms.

In this section

Overview

Introduces the steps in installation.

Installing ILOG CPLEX

Walks through installation steps on different platforms.

Setting up licensing

Explains licensing considerations for ILOG CPLEX.

Using the Component Libraries

Introduces the Component Libraries of ILOG CPLEX.

Overview

You install ILOG CPLEX in two steps: first, install the files from the distribution medium (a CD or an FTP site) into a directory on your local file system; then activate your license.

At that point, all of the features of CPLEX become functional and are available to you. The chapters that follow this one provide tutorials in the use of each of the Technologies that ILOG CPLEX provides: the ILOG Concert Technology Tutorials for C++, Java, and .NET users, and the Callable Library Tutorial for C and other languages.

Important: Please read these instructions in their entirety before you begin the installation. Remember that most ILOG CPLEX distributions will operate correctly only on the specific platform and operating system for which they are designed. If you upgrade your operating system, you may need to obtain a new ILOG CPLEX distribution. In that case, contact your ILOG representative for advice.

Installing ILOG CPLEX

The steps to install ILOG CPLEX involve identifying the correct distribution file for your particular platform, and then executing a command that uses that distribution file. The identification step is explained in the booklet that comes with the CD-ROM, or is provided with the FTP instructions for download. After the correct distribution file is at hand, the installation proceeds as follows.

Installation on UNIX

On UNIX systems ILOG CPLEX is installed in a subdirectory named `cplexXXX`, under the current working directory where you perform the installation.

Use the `cd` command to move to the top level directory into which you want to install the `cplex` subdirectory. Then type either of the following commands:

```
gzip -dc < path/cplex.tgz | tar xf -
```

```
tar xzf path/cplex.tgz
```

where *path* is the full path name pointing to the location of the ILOG CPLEX distribution file (either on the CD-ROM or on a disk where you performed the FTP download). On UNIX systems, both ILOG CPLEX and ILOG Concert Technology are installed when you execute that command.

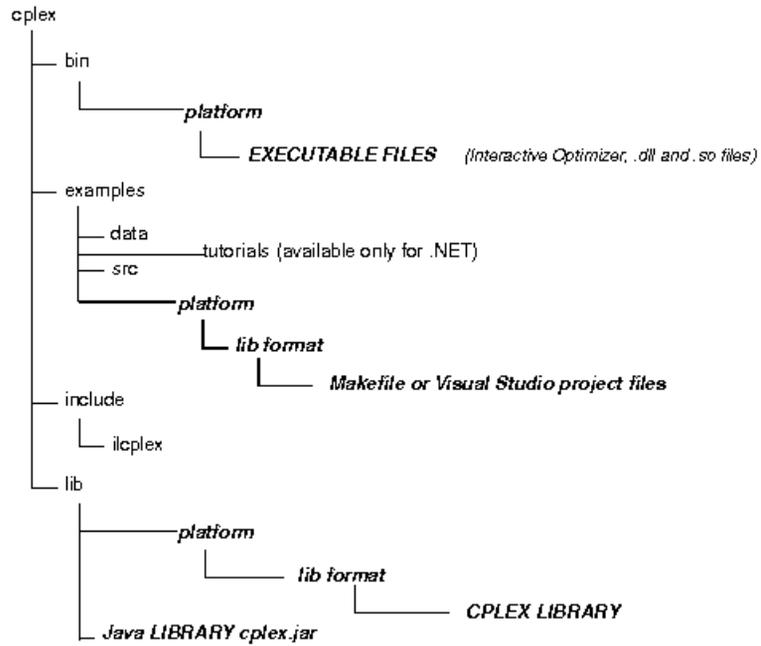
Installation on Windows

Before you install ILOG CPLEX, you need to identify the correct distribution file for your platform. There are instructions on how to identify your distribution in the booklet that comes with the CD-ROM or with the FTP instructions for download. That booklet also tells how to start the ILOG CPLEX installation on your platform.

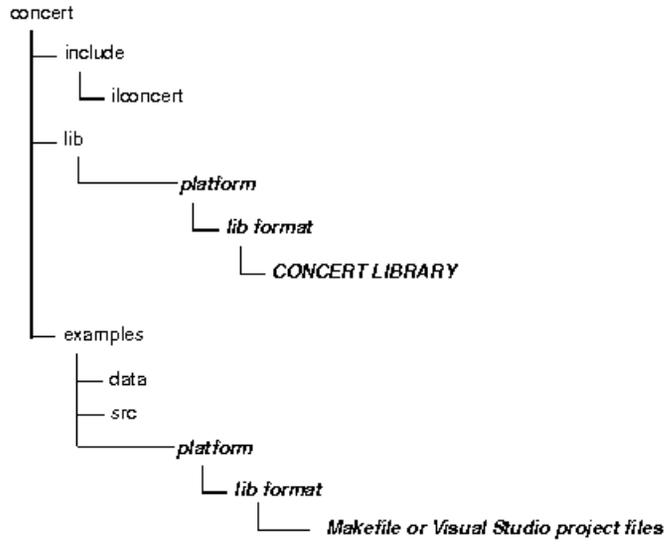
Directory structure

After completing the installation, you will have a directory structure like the one in the illustration *Structure of the ILOG CPLEX installation directory* and *Structure of the Concert Technology Installation Directory*.

Be sure to read the `readme.html` carefully for the most recent information about the version of ILOG CPLEX you have installed.



Structure of the ILOG CPLEX installation directory



Structure of the Concert Technology Installation Directory

Setting up licensing

ILOG CPLEX runs under the control of the ILOG License Manager (ILM). Before you can run ILOG CPLEX, or any application that calls it, you must have established a valid license that ILM can read. Licensing instructions are provided in the *ILOG License Manager User's Guide & Reference*, available from your ILOG customer support website. The basic steps are:

1. Install ILM. Normally you obtain ILM distribution media from the same place that you obtain ILOG CPLEX.
2. Run the `ihostid` program, which is found in the directory where you install ILM.
3. Communicate the output of **Step 2** to your local ILOG sales administration department. They will send you a license key in return. One way to communicate the results of **Step 2** to your local ILOG sales administration department is through the web page serving your region.

Europe and Africa: <https://support.ilog.fr/license/index.cfm>

Americas: <https://support.ilog.com/license/index.cfm>

Asia: <https://support.ilog.com.sg/license/index.cfm>

4. Create a file on your system to hold this license key, and set the environment variable `ILOG_LICENSE_FILE` so that ILOG CPLEX will know where to find the license key. (The environment variable need not be used if you install the license key in a platform-dependent default file location.)

Using the Component Libraries

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

Verifying installation on UNIX

On a UNIX system, go to the subdirectory `examples/ machine / libformat` that matches your particular platform, and in it you will find a file named `Makefile`. Execute one of the examples, for instance `lpex1.c`, by doing

```
make lpex1
```

```
lpex1 -r # this example takes one argument, either -r, -c, or -n
```

If your interest is in running one of the C++ examples, try

```
make ilolpex1
```

```
ilolpex1 -r # this is the same as lpex1 and takes the same arguments.
```

If your interest is in running one of the Java examples, try

Any of these examples should return an optimal objective function value of 202.5.

Verifying installation on Windows

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

To run the examples on Windows, either you must copy the ILOG CPLEX DLL to the directory or folder containing the examples, or you must make sure that the location of the DLL is part of your Windows path.

In case of errors

If an error occurs during the `make` or `compile` step, then check that you are able to access the compiler and the necessary linker/loader files and system libraries. If an error occurs on the next step, when executing the program created by `make`, then the nature of the error message will guide your actions. If the problem is in licensing, consult the ILOG License Manager User's Guide and Reference for further guidance. For Windows users, if the program has trouble locating `cp1ex XXX .dll` or `ILOG.CPLEX.dll`, make sure the DLL is stored either in the current directory or in a directory listed in your `PATH` environment variable.

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

Compiling and linking your own applications

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

- ◆ *Concert Technology tutorial for C++ users* contains information and platform-specific instructions for compiling and linking the Concert Technology Library, for C++ users.
- ◆ *Concert Technology tutorial for Java users* contains information and platform-specific instructions for compiling and linking the Concert Technology Library, for Java users.
- ◆ *Concert Technology tutorial for .NET users* offers an example of a C#.NET application.
- ◆ *Callable Library tutorial* contains information and platform-specific instructions for compiling and linking the Callable Library.

Tutorials

This part provides tutorials to introduce you to each of the components of ILOG CPLEX.

In this section

Solving an LP with ILOG CPLEX

Solves an LP model to contrast ILOG CPLEX components.

Interactive Optimizer tutorial

Introduces the major features of the ILOG CPLEX Interactive Optimizer.

Concert Technology tutorial for C++ users

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

Concert Technology tutorial for Java users

Introduces ILOG CPLEX through ILOG Concert Technology in the Java programming language.

Concert Technology tutorial for .NET users

Introduces ILOG CPLEX through ILOG Concert Technology in the .NET framework.

Callable Library tutorial

Shows how to write applications that use the ILOG CPLEX Callable Library (C API).

Solving an LP with ILOG CPLEX

Solves an LP model to contrast ILOG CPLEX components.

In this section

Overview

Shows ways to solve a linear programming problem.

Problem statement

Displays a linear programming model in a standard formulation to solve in the components of ILOG CPLEX.

Using the Interactive Optimizer

Shows solution of the model in the Interactive Optimizer.

Using Concert Technology in C++

Shows an application to solve the model in the C++ API.

Using Concert Technology in Java

Shows an application to solve the model in the Java API.

Using Concert Technology in .NET

Refers to a tutorial solving the model in the C#.NET API.

Using the Callable Library

Shows an application to solve the model in the C API.

Overview

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

Problem statement

The problem to be solved is:

$$\text{Maximize} \quad x_1 + 2x_2 + 3x_3$$

$$\text{subject to} \quad -x_1 + x_2 + x_3 \leq 20$$

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

$$\text{with these bounds} \quad 0 \leq x_1 \leq 40$$

$$0 \leq x_2 \leq +\infty$$

$$0 \leq x_3 \leq +\infty$$

Using the Interactive Optimizer

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

```
Welcome to CPLEX Interactive Optimizer 11.2.0
  with Simplex, Mixed Integer & Barrier Optimizers
Copyright (c) ILOG 1997-2008
CPLEX is a registered trademark of ILOG

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

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

bounds
0 <= x1 <= 40
0 <= x2
0 <= x3
end
CPLEX> optimize
Tried aggregator 1 time.
No LP presolve or aggregator reductions.
Presolve time =    0.00 sec.

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

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

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

Using Concert Technology in C++

Here is a C++ application using ILOG CPLEX in Concert Technology to solve the example. An expanded form of this example is discussed in detail in *Concert Technology tutorial for C++ users*.

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

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

        IloCplex cplex(model);
        if ( !cplex.solve() ) {
            env.error() << "Failed to optimize LP." << endl;
            throw(-1);
        }

        IloNumArray vals(env);
        env.out() << "Solution status = " << cplex.getStatus() << endl;
        env.out() << "Solution value = " << cplex.getObjValue() << endl;
        cplex.getValues(vals, vars);
        env.out() << "Values = " << vals << endl;
    }
    catch (IloException& e) {
        cerr << "Concert exception caught: " << e << endl;
    }
    catch (...) {
        cerr << "Unknown exception caught" << endl;
    }

    env.end();

    return 0;
}
```

Using Concert Technology in Java

Here is a Java application using ILOG CPLEX with Concert Technology to solve the example. An expanded form of this example is discussed in detail in *Concert Technology tutorial for Java users*.

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

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

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

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

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

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

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

Using Concert Technology in .NET

Here is a C#NET application using Concert Technology with ILOG CPLEX to solve the example. A tutorial offering an expanded version of this application is available in *Concert Technology tutorial for .NET users*.

```
using ILOG.Concert;
using ILOG.CPLEX;
public class Example {
    public static void Main(string[] args) {
        try {
            Cplex cplex = new Cplex();
            double[] lb = {0.0, 0.0, 0.0};
            double[] ub = {40.0, System.Double.MaxValue, System.Double.MaxValue};
            ;

            INumVar[] x = cplex.NumVarArray(3, lb, ub);
            var[0] = x;
            double[] objvals = {1.0, 2.0, 3.0};
            cplex.Add(cplex.Maximize(cplex.ScalProd(x, objvals)));
            rng[0] = new IRange[2];
            rng[0][0] = cplex.AddRange(-System.Double.MaxValue, 20.0);
            rng[0][1] = cplex.AddRange(-System.Double.MaxValue, 30.0);
            rng[0][0].Expr = cplex.Sum(cplex.Prod(-1.0, x[0]),
                                     cplex.Prod( 1.0, x[1]),
                                     cplex.Prod( 1.0, x[2]));
            rng[0][1].Expr = cplex.Sum(cplex.Prod( 1.0, x[0]),
                                     cplex.Prod(-3.0, x[1]),
                                     cplex.Prod( 1.0, x[2]));

            x[0].Name = "x1";
            x[1].Name = "x2";
            x[2].Name = "x3";
            rng[0][0].Name = "c1";
            rng[0][1].Name = "c2";
            cplex.ExportModel("example.lp");
            if ( cplex.Solve() ) {
                double[] x      = cplex.GetValues(var[0]);
                double[] dj     = cplex.GetReducedCosts(var[0]);
                double[] pi     = cplex.GetDuals(rng[0]);
                double[] slack  = cplex.GetSlacks(rng[0]);
                cplex.Output().WriteLine("Solution status = " + cplex.GetStatus());
            };

            cplex.Output().WriteLine("Solution value = " + cplex.ObjValue);
            int nvars = x.Length;
            for (int j = 0; j < nvars; ++j) {
                cplex.Output().WriteLine("Variable   " + j +
                                         ": Value = " + x[j] +
                                         " Reduced cost = " + dj[j]);
            }
            int ncons = slack.Length;
            for (int i = 0; i < ncons; ++i) {
                cplex.Output().WriteLine("Constraint " + i +
```

```
        ": Slack = " + slack[i] +  
        " Pi = " + pi[i]);  
    }  
    }  
    cplex.End();  
}  
catch (ILOG.Concert.Exception e) {  
    System.Console.WriteLine("Concert exception '" + e + "' caught");  
}  
}
```

Using the Callable Library

Here is a C application using the CPLEX Callable Library to solve the example. An expanded form of this example is discussed in detail in *Callable Library tutorial*.

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

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

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

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

    int      solstat;
    double   objval;

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

    lp = CPXcreateprob (env, &status, "lpex1");
    if ( lp == NULL ) {
        fprintf (stderr, "Failed to create LP.\n");
        goto TERMINATE;
    }

    CPXchgobjsen (env, lp, CPX_MAX);
```

```

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

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

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

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

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

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

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

```

TERMINATE:

```

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

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

```

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

return (status);
} /* END main */
```


Interactive Optimizer tutorial

Introduces the major features of the ILOG CPLEX Interactive Optimizer.

In this section

Starting ILOG CPLEX

Explains how to start the Interactive Optimizer.

Using help

Explains how to invoke help in the Interactive Optimizer.

Entering a problem

Documents ways to enter a problem in the Interactive Optimizer.

Displaying a problem

Documents display of a problem in the Interactive Optimizer.

Solving a problem

Documents solving a problem in the Interactive Optimizer.

Performing sensitivity analysis

Describes options to perform sensitivity analysis in the Interactive Optimizer.

Writing problem and solution files

Documents options to write files from the Interactive Optimizer.

Reading problem files

Documents commands and options to read files into the Interactive Optimizer.

Setting ILOG CPLEX parameters

Describes options of the set command to control parameters in the Interactive Optimizer.

Adding constraints and bounds

Describes options to add constraints or bounds to a problem in the Interactive Optimizer.

Changing a problem

Documents commands to change a problem in the Interactive Optimizer.

Executing operating system commands

Describes access to operating system commands from the Interactive Optimizer.

Quitting ILOG CPLEX

Describes the command to terminate a session of the Interactive Optimizer.

Advanced features of the Interactive Optimizer

Suggests topics for further reading about advanced features.

Starting ILOG CPLEX

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

```
cplex
```

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

```
Welcome to CPLEX Interactive Optimizer 11.2.0
  with Simplex, Mixed Integer & Barrier Optimizers
Copyright (c) ILOG 1997-2008
CPLEX is a registered trademark of ILOG

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

CPLEX>
```

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

Using help

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

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

```
CPLEX> help
```

You will see a list of the ILOG CPLEX commands on the screen.

Since all commands start with a unique letter, you could also enter just the single letter `h`.

```
CPLEX> h
```

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

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

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

Enter enough characters to uniquely identify commands & options. Commands can be entered partially (CPLEX will prompt you for further information) or as a

```
whole.
```

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

```
help primopt
```

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

```
h p
```

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

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

```
Syntax: PRIMOPT
```

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

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

The syntax for the `help` command is:

```
help command name
```


Entering a problem

Documents ways to enter a problem in the Interactive Optimizer.

In this section

Overview

Introduces data entry for the Interactive Optimizer.

Entering the example

Describes an example and tells how to enter it in the Interactive Optimizer.

Using the LP format

Describes how to enter a problem in LP format in the Interactive Optimizer.

Entering data

Describes special considerations about entering data from the keyboard.

Overview

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

Entering the example

As an example, this manual uses the following problem:

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

This problem has three variables (x_1 , x_2 , and x_3) and two less-than-or-equal-to constraints.

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

```
enter
```

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

Naming a problem

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

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

The following message appears:

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

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

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

Summary

The syntax for entering a problem is:

```
enter problem name
```

Using the LP format

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

The problem should be entered in the following order:

1. *Objective function*
2. *Constraints*
3. *Bounds*

Objective function

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

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

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

Variable Names

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

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

Constraints

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

```
subject to
```

or

```
st
```

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

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

Constraint Names

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

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

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

Objective Function Names

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

Bounds

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

```
bounds
x1 <= 40
```

Since the bounds on x_2 and x_3 are the same as the default bounds, there is no need to enter them. You have finished entering the problem, so to indicate that the problem is complete, type:

```
end
```

on the last line.

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

Summary

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

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

Entering data

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

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

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

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

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

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

Displaying a problem

Documents display of a problem in the Interactive Optimizer.

In this section

Verifying a problem with the display command

Describes the display command and its options in the Interactive Optimizer.

Displaying problem statistics

Describes options for displaying information about large problems in the Interactive Optimizer.

Specifying item ranges

Describes notation for displaying ranges of items in the Interactive Optimizer.

Displaying variable or constraint names

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

Ordering variables

Describes internal order among variables in the Interactive Optimizer.

Displaying constraints

Describes options to display constraints in the Interactive Optimizer.

Displaying the objective function

Describes options to display an objective function in the Interactive Optimizer.

Displaying bounds

Describes options to display bounds of a problem in the Interactive Optimizer.

Displaying a histogram of nonzero counts

Describes options to display summary of nonzero rows and columns in the Interactive Optimizer.

Verifying a problem with the display command

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

```
display
```

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

```
Display Options:
```

```
conflict      display conflict that demonstrates model infeasibility
problem       display problem characteristics
sensitivity   display sensitivity analysis
settings      display parameter settings
solution      display existing solution
```

```
Display what:
```

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

```
Display Problem Options:
```

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

```
Display which problem characteristic:
```

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

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

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

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

Summary

Display problem characteristics by entering the command:

```
display problem
```

Displaying problem statistics

When the problem is as small as our example, it is easy to display it on the screen; however, many real problems are far too large to display. For these problems, the `stats` option of the `display problem` command is helpful. When you select `stats`, information about the attributes of the problem appears, but not the entire problem itself. These attributes include:

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

Try this feature by typing:

```
display problem stats
```

For our example, the following information appears:

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

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

```
Variables      : Min LB: 0.000000      Max UB: 40.000000
Objective nonzeros : Min   : 1.000000      Max   : 3.000000
Linear constraints :
  Nonzeros      : Min   : 1.000000      Max   : 3.000000
  RHS nonzeros   : Min   : 20.000000     Max   : 30.000000
```

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

- ◆ `names`, documented in *Displaying variable or constraint names*, can be used to display a specified set of variable or constraint names;

- ◆ `constraints` , documented in *Displaying constraints*, can be used to display a specified set of constraints;
- ◆ `bounds` , documented in *Displaying bounds*, can be used to display a specified set of bounds.

Specifying item ranges

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

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

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

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

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

Displaying variable or constraint names

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

Type the following command:

```
display problem names variables
```

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

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

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

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

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

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

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

```
x1  x2  x3
```

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

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

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

```
display problem names variables
```

```
Display which variable name(s): 2  
x2
```

Summary

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

```
display problem names variables
```

- ◆ You can display constraint names by entering the command:

```
display problem names constraints
```

Ordering variables

In the example problem there is a direct correlation between the variable names and their numbers (x_1 is variable 1, x_2 is variable 2, etc.); that is not always the case. The internal ordering of the variables is based on their order of occurrence when the problem is entered. For example, if x_2 had not appeared in the objective function, then the order of the variables would be x_1 , x_3 , x_2 .

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

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

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

Displaying constraints

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

```
display problem constraints 2
```

The second constraint appears:

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

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

```
display problem constraints *
```

Displaying the objective function

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

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

Displaying bounds

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

```
display problem bounds -
```

or, try a wildcard, like this:

```
display problem bounds *
```

The result is:

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

Summary

The general syntax of the `display` command is:

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

Displaying a histogram of nonzero counts

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

```
display problem histogram c
```

```
display problem histogram r
```

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

```
Column counts (excluding fixed variables):  
    Nonzero Count:    2  
Number of Columns:    3
```

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

```
Row counts (excluding fixed variables):  
    Nonzero Count:    3  
Number of Rows:      2
```

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

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

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

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

Solving a problem

Documents solving a problem in the Interactive Optimizer.

In this section

Overview

Introduces solving a model and displaying its solution in the Interactive Optimizer.

Solving the example

Describes activity of the Interactive Optimizer during solution of a problem.

Solution options

Describes additional options after solving in the Interactive Optimizer.

Displaying post-solution information

Describes display options for post-solution information in the Interactive Optimizer.

Overview

If you have been following this tutorial step by step, the problem is now correctly entered, and you can now use ILOG CPLEX to solve it. This tutorial continues with the following topics, covering solving the problem and displaying solution information.

Solving the example

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

Entering the optimize command

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

```
optimize
```

Preprocessing

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

Monitoring the iteration log

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

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

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

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

Reporting the solution

After it finds the optimal solution, ILOG CPLEX reports:

- ◆ the objective function value
- ◆ the problem solution time in seconds

- ◆ the total iteration count
- ◆ the Phase I iteration count (in parentheses)

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

```
Tried aggregator 1 time.  
No LP presolve or aggregator reductions.  
Presolve Time = 0.00 sec.  
  
Iteration Log . . .  
Iteration:    1  Dual infeasibility =           0.000000  
Iteration:    2  Dual objective      =           202.500000  
  
Dual simplex - Optimal: Objective =    2.0250000000e+02  
Solution Time =    0.00 sec. Iterations = 2 (1)  
  
CPLEX>
```

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

Summary

To solve an LP problem, use the command:

```
optimize
```

Solution options

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

- ◆ *Filing iteration logs;*
- ◆ *Re-solving;*
- ◆ *Using alternative optimizers;*
- ◆ *Interrupting the optimization.*

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

Filing iteration logs

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

Re-solving

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

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

Using alternative optimizers

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

Interrupting the optimization

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

Displaying post-solution information

After an optimal solution is found, ILOG CPLEX can provide many different kinds of information for viewing and analyzing the results. This information is accessed via the `display` command and via some `write` commands.

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

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

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

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

```
display solution variables -
```

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

Variable Name	Solution Value
x1	40.000000
x2	17.500000
x3	42.500000

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

```
display solution slacks -
```

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

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

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

```
display solution dual -
```

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

Constraint Name	Dual Price
c1	2.750000
c2	0.250000

Summary

Display solution characteristics by entering a command with the syntax:

```
display solution identifier
```

Performing sensitivity analysis

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

Sensitivity analysis can be performed on the following:

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

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

```
display sensitivity obj -
```

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

```
display sensitivity obj *
```

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

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

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

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

```
display sensitivity rhs -
```

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

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

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

ILOG CPLEX can also display lower bound sensitivity ranges with the command

```
display sensitivity lb
```

and upper bound sensitivity with the command

```
display sensitivity ub
```

Summary

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

```
display sensitivity identifier
```

Writing problem and solution files

Documents options to write files from the Interactive Optimizer.

In this section

Overview

Introduces the write command of the Interactive Optimizer.

Selecting a write file format

Describes formats available to write a file from the Interactive Optimizer.

Writing LP files

Describes the command for LP file format from the Interactive Optimizer.

Writing basis files

Describes the command to write basis files from the Interactive Optimizer.

Using path names

Describes special considerations with respect to path names from the Interactive Optimizer.

Overview

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

Selecting a write file format

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

```
File type options:

bas      INSERT format basis file
clp      Conflict file
dpe      Binary format for dual-perturbed problem
dua      MPS format of explicit dual of problem
emb      MPS format of (embedded) network
flt      Solution pool filters
lp       LP format problem file
min      DIMACS min-cost network-flow format of (embedded) network
mps      MPS format problem file
mst      MIP start file
net      CPLEX network format of (embedded) network
ord      Integer priority order file
ppe      Binary format for primal-perturbed problem
pre      Binary format for presolved problem
prm      Non-default parameter settings
rlp      LP format problem with generic names
rew      MPS format problem with generic names
sav      Binary matrix and basis file
sol      Solution file

File type:
```

- ◆ The BAS format is used for storing basis information and is introduced in *Writing basis files*. See also *Reading basis files*.
- ◆ The LP format was discussed in *Using the LP format*. Using this format is explained in *Writing LP files* and *Reading LP files*.
- ◆ The MPS format is covered in *Reading MPS files*.

Note: All these file formats are documented in more detail in the reference manual *ILOG CPLEX File Formats*.

Writing LP files

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

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

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

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

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

```
write example2 lp
```

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

```
write example.lp
```

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

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

```
write example.lp mps
```

Writing basis files

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

```
write example.bas
```

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

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

Using path names

A full path name may also be included to indicate on which drive and directory any file should be saved. The following might be a valid `write` command if the disk drive on your system contains a root directory named `problems` :

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

Summary

The general syntax for the `write` command is:

```
write filename file_format
```

or

```
write filename.file_extension
```

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

Reading problem files

Documents commands and options to read files into the Interactive Optimizer.

In this section

Overview

Describes file formats and conventions for using them in the Interactive Optimizer.

Selecting a read file format

Describes files formats that the Interactive Optimizer reads.

Reading LP files

Describes the command to read a formatted LP file into the Interactive Optimizer.

Using file extensions

Describes conventions governing file extensions in the Interactive Optimizer.

Reading MPS files

Describes options to read MPS formatted files in the Interactive Optimizer.

Reading basis files

Describes options to read basis files into the Interactive Optimizer.

Overview

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

Selecting a read file format

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

```
File type options:

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

File type:
```

Note: All these file formats are documented in more detail in the reference manual *ILOG CPLEX File Formats*.

Reading LP files

At the CPLEX> prompt type:

```
read
```

The following message appears requesting a file name:

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

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

```
example
```

```
example2
```

```
example.lp
```

```
example.bas
```

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

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

```
lp
```

ILOG CPLEX displays a confirmation message, like this:

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

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

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

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

Using file extensions

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

```
read example.lp
```

Reading MPS files

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

```
read afiro mps
```

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

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

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

Reading basis files

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

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

```
read example.bas
```

The message of confirmation:

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

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

Summary

The general syntax for the `read` command is:

```
read filename file_format
```

or

```
read filename.file_extension
```

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

Setting ILOG CPLEX parameters

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

Setting a parameter

To see the parameters that can be changed, type:

```
set
```

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

```
Available Parameters:
advance          set indicator for advanced starting information
barrier          set parameters for barrier optimization
clocktype        set type of clock used to measure time
conflict         set parameters for finding conflicts
defaults         set all parameter values to defaults
emphasis         set optimization emphasis
feasopt          set parameters for feasoit
logfile          set file to which results are printed
lpmethod         set method for linear optimization
mip              set parameters for mixed integer optimization
network          set parameters for network optimizations
output           set extent and destinations of outputs
parallel         set parallel optimization mode
preprocessing    set parameters for preprocessing
qpmethod         set method for quadratic optimization
read             set problem read parameters
sifting          set parameters for sifting optimization
simplex           set parameters for primal, dual simplex optimizations
threads          set default parallel thread count
timelimit        set time limit in seconds
tune             set parameters for parameter tuning
workdir          set directory for working files
workmem          set memory available for working storage (megabytes)

Parameter to set:
```

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

```
No parameters changed.
```

Resetting defaults

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

```
set defaults
```

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

Summary

The general syntax for the `set` command is:

```
set parameter option new_value
```

Displaying parameter settings

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

```
display settings all
```

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

```
display settings changed
```

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

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

Adding constraints and bounds

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

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

$$x_1 + 2x_2 + 3x_3 \geq 50$$

You may do either interactively or from a file.

Adding interactively

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

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

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

```
display problem all

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

Adding from a file

Alternatively, you may read in new constraints and bounds from a file. If you enter a file name after the `add` command, ILOG CPLEX will read a file matching that name. The file contents

must comply with standard ILOG CPLEX LP format. ILOG CPLEX does not prompt for a file name if none is entered. Without a file name, interactive entry is assumed.

Summary

The general syntax for the `add` command is:

`add`

or

`add filename`

Changing a problem

Documents commands to change a problem in the Interactive Optimizer.

In this section

Overview

Introduces the change command for modifying a model in the Interactive Optimizer.

What can be changed?

Describes the options of the change command.

Changing constraint or variable names

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

Changing sense

Describes the option to change the sense of a constraint in the Interactive Optimizer.

Changing bounds

Describes the option to change the bounds of a variable in the Interactive Optimizer.

Removing bounds

Describes the way to remove a bound in the Interactive Optimizer.

Changing coefficients of variables

Describes the means to change the coefficient of a variable in a constraint in the Interactive Optimizer.

Objective and RHS coefficients

Describes the means to change a coefficient in the objective function in the Interactive Optimizer.

Deleting entire constraints or variables

Describes the options of the delete command in the Interactive Optimizer.

Changing small values to zero

Describes a means to clean data by zeroing out small values in the Interactive Optimizer.

Overview

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

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

What can be changed?

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

```
change
```

The elements that can be changed are displayed like this:

```
Change options:
```

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

```
Change to make:
```

Changing constraint or variable names

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

```
Change to make: name
```

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

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

The name of the constraint has been changed.

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

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

Changing sense

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

```
change sense
```

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

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

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

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

The sense of the constraint has been changed.

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

Changing bounds

When the example was entered, bounds were set specifically only for the variable x_1 . The bounds can be changed on this or other variables with the `bounds` option. Again, start by selecting the command and option.

```
change bounds
```

Select the variable by name or number and then select which bound you would like to change. For the example, change the upper bound of variable x_2 from $+$ to 50.

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

Removing bounds

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

```
change bounds x2 u inf
```

You receive the message:

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

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

Changing coefficients of variables

Up to this point all of the changes that have been made could be referenced by specifying a single constraint or variable. In changing a coefficient, however, a constraint *and* a variable must be specified in order to identify the correct coefficient. As an example, change the coefficient of x_3 in the `new3` constraint from 3 to 30.

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

```
change coefficient
```

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

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

The final step is to enter the new value for the coefficient of x_3 .

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

Objective and RHS coefficients

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

```
change rhs 1 25.0
```

Deleting entire constraints or variables

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

```
change delete
```

ILOG CPLEX displays a list of delete options.

```
Delete options:
```

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

```
Deletion to make:
```

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

```
Deletion to make: constraints
```

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

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

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

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

```
display problem all
Maximize
```

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

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

Changing small values to zero

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

```
change values
```

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

Summary

The general syntax for the change command is:

```
change option identifier [identifier2] new value
```

Executing operating system commands

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

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

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

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

Summary

The general syntax for the `execute` command is:

```
execute command line
```

Quitting ILOG CPLEX

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

```
quit
```

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

Advanced features of the Interactive Optimizer

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

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

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

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

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

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

Concert Technology tutorial for C++ users

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

In this section

The design of CPLEX in Concert Technology C++ applications

Explains objects necessary to an application of ILOG CPLEX in C++.

Compiling ILOG CPLEX in Concert Technology C++ applications

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

The anatomy of an ILOG Concert Technology C++ application

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

Building and solving a small LP model in C++

Shows a sample solving a linear programming model in C++.

Writing and reading models and files

Introduces reading of models from files and writing models to files in a C++ application.

Selecting an optimizer

Outlines criteria for selecting an optimizer in a C++ application.

Reading a problem from a file: example ilolpex2.cpp

Introduces the sample ilolpex2.cpp to illustrate reading from a file.

Modifying and re-optimizing

Walks through modification of the model and re-optimizing in the sample.

Modifying an optimization problem: example ilolpex3.cpp

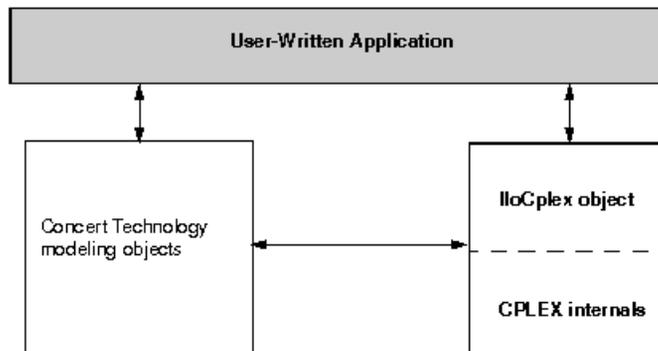
Introduces a sample to illustrate how to modify a model.

The design of CPLEX in Concert Technology C++ applications

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

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

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



A View of ILOG CPLEX with ILOG Concert Technology

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

This chapter gives a brief tutorial illustrating the modeling and solution classes provided by ILOG Concert Technology and ILOG CPLEX. More information about the algorithm class `IloCplex` and its nested classes can be found in the *ILOG CPLEX User's Manual* and *ILOG CPLEX Reference Manual*.

Compiling ILOG CPLEX in Concert Technology C++ applications

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

In this section

Testing your installation on UNIX

Suggests ways to test installation of ILOG CPLEX on a UNIX platform.

Testing your installation on Windows

Suggests ways to test installation of ILOG CPLEX on a Windows platform.

In case of problems

Recommends trouble-shooting procedures.

Testing your installation on UNIX

To run the test, follow these steps.

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

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

`make all_cpp` compiles and links the examples of the C++ API.

4. Execute one of the compiled examples.

`make execute_all` executes all of the examples.

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

Testing your installation on Windows

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

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

In case of problems

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

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

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

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

If you get a message from the linker, such as

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

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

If you get a message such as

```
ilm: CPLEX: no license found for this product
```

or

```
ilm: CPLEX: invalid encrypted key "MNJVUXTDJV82" in "/usr/ilog/  
ilm/access.ilm";run ilmcheck
```

then there is a problem with your license to use ILOG CPLEX. Review the *ILOG License Manager User's Guide and Reference* to see whether you can correct the problem. If not, contact the customer support hotline and repeat the error message there.

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

The anatomy of an ILOG Concert Technology C++ application

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

In this section

Constructing the environment: IloEnv

Describes the class IloEnv.

Creating a model: IloModel

Describes modeling objects for an application of ILOG CPLEX in C++.

Solving the model: IloCplex

Describes the class IloCplex.

Querying results

Introduces query methods.

Handling errors

Suggests way to handle errors in an application of ILOG CPLEX in C++.

Constructing the environment: IloEnv

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

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

```
IloEnv env;
```

Note: The environment object created in an ILOG Concert Technology application is different from the environment created in the ILOG CPLEX C library by calling the routine `CPXopenCPLEXCPXopenCPLEX`.

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

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

```
IloEnv env2 = env;
```

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

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

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

Creating a model: IloModel

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

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

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

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

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

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

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

<code>IloNumVar</code>	representing modeling variables;
<code>IloRange</code>	defining constraints of the form $l \leq \text{expr} \leq u$, where <code>expr</code> is a linear expression; and
<code>IloObjective</code>	representing an objective function.

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

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

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

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

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

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

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

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

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

This creates the extractable `obj` of type `IloObjective` which represents the objective function of the example presented in *Introducing ILOG CPLEX*.

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

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

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

This way there is no need for the program variable `obj` and the program is shorter. If in contrast, the objective function is needed later, for example, to change it and reoptimize the model when doing scenario analysis, the variable `obj` must be created in order to refer to the

objective function. (From the standpoint of algorithmic efficiency, the two approaches are comparable.)

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

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

The part $-x_1 + x_2 + x_3 \leq 20$ creates an object of class `IloRange` that is immediately added to the model by passing it to the method `IloModel::add`. Again, if a reference to the `IloRange` object is needed later, an `IloRange` handle object must be stored for it. Concert Technology provides flexible array classes for storing data, such as these `IloRange` objects. As with variables, Concert Technology provides a variety of constructors that help create range constraints.

While those examples use expressions with modeling variables directly for modeling, it should be pointed out that such expressions are themselves represented by yet another Concert Technology class, `IloExpr`. Like most Concert Technology objects, `IloExpr` objects are handles. Consequently, the method `end` must be called when the object is no longer needed. The only exceptions are implicit expressions, where the user does not create an `IloExpr` object, such as when writing (for example) $x_1 + 2 * x_2$. For such implicit expressions, the method `end` should not be called. The importance of the class `IloExpr` becomes clear when expressions can no longer be fully spelled out in the source code but need instead to be built up in a loop. Operators like `+=` provide an efficient way to do this.

Solving the model: IloCplex

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

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

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

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

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

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

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

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

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

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

Querying results

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

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

For example:

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

stores the solution value for the modeling variable x_1 in `val1`. Other methods are available for querying other solution information. For example, the objective function value of the solution can be accessed using:

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

Handling errors

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

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

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

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

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

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

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

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

Building and solving a small LP model in **C++**

Shows a sample solving a linear programming model in C++.

In this section

Overview

Introduces three alternative approaches in the sample.

Modeling by rows

Describes the approach of modeling by rows in the sample.

Modeling by columns

Describes the approach of modeling by columns in the sample.

Modeling by nonzero elements

Describes the approach of modeling by nonzero elements in the sample.

Overview

A complete example of building and solving a small LP model can now be presented. This example demonstrates:

- ◆ *Modeling by rows*
- ◆ *Modeling by columns*
- ◆ *Modeling by nonzero elements*

Example `ilolplex1.cpp`, which is one of the example programs in the standard ILOG CPLEX distribution, is an extension of the example presented in *Introducing ILOG CPLEX*. It shows three different ways of creating an ILOG Concert Technology LP model, how to solve it using `IloCplex`, and how to access the solution. Here is the problem that the example optimizes:

```
Maximize       $x_1 + 2x_2 + 3x_3$ 

subject to     $-x_1 + x_2 + x_3 \leq 20$ 
               $x_1 - 3x_2 + x_3 \leq 30$ 

with these bounds   $0 \leq x_1 \leq 40$ 
                   $0 \leq x_2 \leq$ 
                   $0 \leq x_3 \leq$ 
```

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

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

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

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

If a solution is found, solution information is output through the channel, `env.out` which is initialized to `cout` by default. The output operator `<<` is defined for type `IloAlgorithm::Status` as returned by the call to `getStatus`. It is also defined for `IloNumArray`, the ILOG Concert Technology class for an array of numerical values, as returned by the calls to `getValues`, `getDuals`, `getSlacks`, and `getReducedCosts`. In general, the output operator is defined for any ILOG Concert Technology array of elements if the output operator is defined for the elements.

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

Modeling by rows

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

Modeling by columns

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

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

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

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

The column expression is passed as an argument to the constructor of class `IloNumVar`. For example the constructor `IloNumVar(obj(1.0) + c[0](-1.0) + c[1](1.0), 0.0, 40.0)` creates a new modeling variable with lower bound 0.0, upper bound 40.0 and, by default, type `ILOFLOAT`, and adds it to the objective `obj` with a linear coefficient of 1.0, to the range `c[0]` with a linear coefficient of -1.0 and to `c[1]` with a linear coefficient of 1.0. Column expressions can be used directly to construct numerical variables with default bounds `[0, IloInfinity]` and type `ILOFLOAT`, as in the following statement:

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

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

Modeling by nonzero elements

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

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

Writing and reading models and files

In example `ilo1pex1.cpp`, one line is still unexplained:

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

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

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

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

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

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

and

```
void IloCplex::importModel (IloModel& m,  
                           const char* filename,
```

```
IloObjective& obj,  
IloNumVarArray vars,  
IloRangeArray rngs,  
IloSOS1Array sos1,  
IloSOS2Array sos2) const;
```

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

Selecting an optimizer

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

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

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

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

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

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

The optimizer option used for solving pure LPs and QPs is controlled by setting the root algorithm argument. This is demonstrated next, in example `ilolpex2.cpp`.

Reading a problem from a file: example ilolpex2.cpp

Introduces the sample `ilolpex2.cpp` to illustrate reading from a file.

In this section

Overview

Outlines the sample.

Reading the model from a file

Walks through reading a model from a file in the sample.

Selecting the optimizer

Walks through selecting the optimizer in the sample.

Accessing basis information

Walks through access to basis information in the sample.

Querying quality measures

Walks through queries for measurement of the quality of a solution in the sample.

Overview

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

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

```
ilolpex2 example.mps d
```

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

Example `ilolpex2` demonstrates:

- ◆ *Reading the model from a file*
- ◆ *Selecting the optimizer*
- ◆ *Accessing basis information*
- ◆ *Querying quality measures*

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

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

Reading the model from a file

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

Selecting the optimizer

The selection of the optimizer option is done in the switch statement controlled by the second command line argument, a parameter. A call to `setParam(IloCplex::RootAlg, alg)` selects the desired `IloCplex::Algorithm` option.

Accessing basis information

After solving the model by calling the method `solve`, the results are accessed in the same way as in `ilolpex1.cpp`, with the exception of basis information for the variables. It is important to understand that not all optimizer options compute basis information, and thus it cannot be queried in all cases. In particular, basis information is not available when the model is solved using the barrier optimizer (`IloCplex::Barrier`) without crossover (parameter `IloCplex::BarCrossAlg` set to `IloCplex::NoAlg`).

Querying quality measures

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

Modifying and re-optimizing

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

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

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

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

Modifying an optimization problem: example ilolpex3.cpp

Introduces a sample to illustrate how to modify a model.

In this section

Overview

Outlines the sample.

Setting ILOG CPLEX parameters

Walks through setting parameters in the sample.

Modifying an optimization problem

Walks through modification of the model in the sample.

Starting from a previous basis

Walks through starting from a basis in the sample.

Complete program

Points to online version of the sample.

Overview

This example demonstrates:

- ◆ *Setting ILOG CPLEX parameters*
- ◆ *Modifying an optimization problem*
- ◆ *Starting from a previous basis*

Here is the problem example `ilolpex3` solves:

$$\begin{aligned} &\text{Minimize} && c^T x \\ &\text{subject to} && Hx = d \\ & && Ax = b \\ & && l \leq x \leq u \end{aligned}$$

where

$$\begin{aligned} H = & \begin{pmatrix} -1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & -1 \end{pmatrix} & d = \begin{pmatrix} -3 \\ 1 \\ 4 \\ 3 \\ -5 \end{pmatrix} \\ A = & \begin{pmatrix} 2 & 1 & -2 & -1 & 2 & -1 & -2 & -3 \\ 1 & -3 & 2 & 3 & -1 & 2 & 1 & 1 \end{pmatrix} & b = \begin{pmatrix} 4 \\ -2 \end{pmatrix} \\ c = & (-9 \ 1 \ 4 \ 2 \ -8 \ 2 \ 8 \ 12) \\ l = & (0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0) \\ u = & (50 \ 50 \ 50 \ 50 \ 50 \ 50 \ 50 \ 50) \end{aligned}$$

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

1. The ILOG CPLEX Network Optimizer is used to solve

Minimize $c^T x$
 subject to $Hx = d$
 $l \leq x \leq u$

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

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

Before the model is solved, the network optimizer is selected by setting the `RootAlg` parameter to the value `IloCplex::Network`, as shown in example `ilolplex2.cpp`. The simplex display parameter `SimDisplay` is set so that the simplex algorithm issues logging information as it executes.

Setting ILOG CPLEX parameters

`IloCplex` provides a variety of parameters that allow you to control the solution process. They can be categorized as Boolean, integer, numeric, and string parameters and are represented by the enumeration types `IloCplex_BoolParam`, `IloCplex_IntParam`, `IloCplex_NumParam`, and `IloCplex_StringParam`, respectively.

Modifying an optimization problem

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

Starting from a previous basis

Before solving the modified problem, example `il01pex3.cpp` sets the optimizer option to `Dual`, as this is the algorithm that can generally take best advantage of the optimal basis from the previous solve after the addition of constraints.

Complete program

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

Concert Technology tutorial for Java users

Introduces ILOG CPLEX through ILOG Concert Technology in the Java programming language.

In this section

Overview

Highlights procedures in a typical application with ILOG CPLEX in Java.

Compiling ILOG CPLEX in ILOG Concert Technology Java applications

Explains how to compile ILOG CPLEX in a Java application.

The design of ILOG CPLEX in ILOG Concert Technology Java applications

Describes ILOG CPLEX as a component in a Java application.

The anatomy of an ILOG Concert Technology Java application

Outlines steps in a Java application of ILOG CPLEX.

Building and solving a small LP model in Java

Uses an example to show how to solve a model in a Java application of ILOG CPLEX.

Overview

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

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

Compiling ILOG CPLEX in ILOG Concert Technology Java applications

Explains how to compile ILOG CPLEX in a Java application.

In this section

Overview

Introduces general considerations about ILOG CPLEX in Java applications.

Adapting build procedures to your platform

Introduces makefiles and other aids to support Java application development with ILOG CPLEX.

In case problems arise

Suggests trouble-shooting procedures specific to Java applications.

Overview

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

```
-classpath path_to_cplex.jar
```

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

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

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

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

Adapting build procedures to your platform

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

Here are suggestions for establishing build procedures for your application.

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

In case problems arise

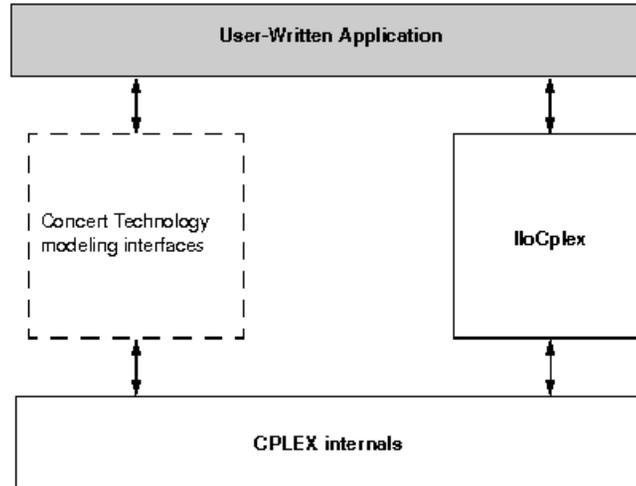
If a problem occurs in the compilation phase, make sure your java compiler is correctly set up and that your classpath includes the `cplex.jar` file.

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

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

```
-Djava.library.path=<path_to_shared_library>
```
3. If you get a message like `ilm:CPLEX: no license found for this product or ilm: CPLEX: invalid encrypted key "MNJVUXTDJ82"` in `"/usr/ilog/ilm/ access.ilm"` run `ilmcheck`, then there is a problem with your license to use ILOG CPLEX. Review the *ILOG License Manager User's Guide and Reference* to see whether you can correct the problem. If you have verified your system and license setup but continue to experience problems, contact ILOG customer support and report the error messages.

The design of ILOG CPLEX in ILOG Concert Technology Java applications



A View of ILOG CPLEX in ILOG Concert Technology

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

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

The anatomy of an ILOG Concert Technology Java application

Outlines steps in a Java application of ILOG CPLEX.

In this section

Overview

Explains object-oriented considerations for Java applications using ILOG CPLEX.

Create the model

Explains how to create a model in a Java application of ILOG CPLEX.

Solve the model

Explains how to create the object-oriented optimizer in a Java application of ILOG CPLEX.

Query the results

Describes how to query results from a Java application of ILOG CPLEX.

Overview

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

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

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

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

```
-classpath <path_to_cplex.jar>

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

Create the model

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

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

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

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

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

can be created like this:

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

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

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

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

Every modeling object that is to be used in a model must be added to the `IloCplex` object. The variables need not be explicitly added as they are treated implicitly when used in the

expression of the objective. More generally, every modeling object that is referenced by another modeling object which itself has been added to `IloCplex`, is implicitly added to `IloCplex` as well.

There is a shortcut notation for creating and adding the objective to `addMinimize()` :

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

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

$$-x[0] + x[1] + x[2] \leq 20.0$$

can be added by calling:

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

Solve the model

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

After you have created a model as explained in *Create the model*, the `IloCplex` object `cplex` is ready to solve the problem, which consists of the model and all the modeling objects that have been added to it. Invoking the optimizer then is as simple as calling the method `solve()`.

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

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

Query the results

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

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

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

Building and solving a small LP model in Java

Uses an example to show how to solve a model in a Java application of ILOG CPLEX.

In this section

Overview

Introduces the example to illustrate solving a model in a Java application of ILOG CPLEX.

Modeling by rows

Demonstrates modeling by rows in the example of a Java application of ILOG CPLEX.

Modeling by columns

Demonstrates modeling by columns in the example of a Java application of ILOG CPLEX.

Modeling by nonzeros

Demonstrates modeling by nonzeros in the example of a Java application of ILOG CPLEX.

Overview

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

- ◆ *Modeling by rows;*
- ◆ *Modeling by columns;*
- ◆ *Modeling by nonzeros.*

Example `LPex1.java` is an extension of the example presented in *Entering the example*:

```
Maximize       $x_1 + 2x_2 + 3x_3$ 

subject to     $-x_1 + x_2 + x_3 \leq 20$ 
               $x_1 - 3x_2 + x_3 \leq 30$ 

with these bounds   $0 \leq x_1 \leq 40$ 
                   $0 \leq x_2 \leq$ 
                   $0 \leq x_3 \leq$ 
```

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

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

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

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

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

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

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

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

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

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

Modeling by rows

The method `populateByRow` creates the model by adding the finished constraints and objective function to the active model, one by one. It does so by first creating the variables with the method `cplex.numVarArray(ilog.concert.IloColumnArray cols, double[] lb, double[] ub)`. Then the minimization objective function is created and added to the active model with the method `IloCplex.addMinimize()`. The expression that defines the objective function is created by a method, `IloCplex.scalarProd(double[] vals, ilog.concert.IloNumVar[] vars)`, that forms a scalar product using an array of objective coefficients times the array of variables. Finally, each of the two constraints of the model are created and added to the active model with the method `IloCplex.addLe(double v, ilog.concert.IloNumExpr e)`. For building the constraint expression, the methods `IloCplex.sum(double v, ilog.concert.IloNumExpr e1)` and `IloCplex.prod(double v, ilog.concert.IloNumExpr e1)` are used, as a contrast to the approach used in constructing the objective function.

Modeling by columns

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

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

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

An instance of `IloColumn` created with the method `IloCplex`.

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

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

The three methods `model.column` create `IloColumn` objects for installing a new variable in the objective `obj` and in the constraints `r0` and `r1`, with linear coefficients `1.0`, `-1.0`

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

Modeling by nonzeros

The last of the three functions for building the model is `populateByNonzero`. This function creates the variables with only their bounds, and the empty constraints, that is, ranged constraints with only lower and upper bound but with no expression. Only after that are the expressions constructed over these existing variables, in a manner similar to the ones already described; they are installed in the existing constraints with the method `IloRange.setExpr(iilog.concert.IloNumExpr expr)`.

Concert Technology tutorial for .NET users

Introduces ILOG CPLEX through ILOG Concert Technology in the .NET framework.

In this section

Presenting the tutorial

Introduces a tutorial for users of ILOG CPLEX in .NET framework.

What you need to know: prerequisites

Outlines prerequisites for the tutorial.

What you will be doing

Outlines activities of the tutorial.

Describe

Outlines questions to ask in order to describe an optimization problem adequately for application development.

Model

Outlines steps toward building a model for the problem.

Solve

Outlines steps for adding the parts of the application that solve the problem.

Complete program

Points to an online version of the problem.

Presenting the tutorial

This tutorial introduces ILOG CPLEX through ILOG Concert Technology in the .NET framework. It gives you an overview of a typical application and highlights procedures for:

- ◆ Creating a model
- ◆ Populating the model with data, either by rows, by columns, or by nonzeros
- ◆ Solving that model
- ◆ Displaying results after solving

This chapter concentrates on an example using C#.NET. There are also examples of VB.NET (Visual Basic in the .NET framework) delivered with ILOG CPLEX in *yourCPLEXhome\examples\src\vb*. Because of their .NET framework, those VB.NET examples differ from the traditional Visual Basic examples that may already be familiar to some ILOG CPLEX users.

In the standard distribution of the product, the file `dotnet.html` offers useful details about installing the product as well as compiling and executing examples.

Note: This 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 Microsoft Visual Studio.

What you need to know: prerequisites

This tutorial requires a working knowledge of C#.NET.

If you are experienced in mathematical programming or operations research, you are probably already familiar with many concepts used in this tutorial. However, little or no experience in mathematical programming or operations research is required to follow this tutorial.

You should have ILOG CPLEX and ILOG Concert Technology for .NET users **installed** in your development environment before starting this tutorial. In your integrated development environment, you should be able to **compile**, **link**, and **execute** a sample application provided with ILOG CPLEX and ILOG Concert Technology for .NET users before starting the tutorial.

To check your installation before starting the tutorial, open

```
yourCPLEXhome \examples\ platform \ format \examples.net.sln
```

in your integrated development environment, where *yourCPLEXhome* indicates the place you installed ILOG CPLEX on your platform, and *format* indicates one of these possibilities: *stat_mda*, *stat_mta*, or *stat_sta*. Your integrated development environment, Microsoft Visual Studio, will then check for the DLLs of ILOG CPLEX and ILOG Concert Technology for .NET users and warn you if they are not available to it.

Another way to check your installation is to load the project for one of the samples delivered with your product. For example, you might load the following project into Microsoft Visual Studio to check a C# example of the diet problem:

```
yourCPLEXhome \examples\ platform \ format \Diet.csproj
```

What you will be doing

ILOG CPLEX can work together with ILOG Concert Technology for .NET users, a .NET library that allows you to model optimization problems independently of the algorithms used to solve the problem. It provides an extensible modeling layer adapted to a variety of algorithms ready to use off the shelf. This modeling layer enables you to change your model, without completely rewriting your application.

To find a solution to a problem by means of ILOG CPLEX with ILOG Concert Technology for .NET users, you use a three-stage method: describe, model, and solve.

The first stage is to describe the problem in natural language.

The second stage is to use the classes and interfaces of ILOG Concert Technology for .NET users to model the problem. The model is composed of data, decision variables, and constraints. Decision variables are the unknown information in a problem. Each decision variable has a domain of possible values. The constraints are limits or restrictions on combinations of values for these decision variables. The model may also contain an objective, an expression that can be maximized or minimized.

The third stage is to use the classes of ILOG Concert Technology for .NET users to solve the problem. Solving the problem consists of finding a value for each decision variable while simultaneously satisfying the constraints and maximizing or minimizing an objective, if one is included in the model.

In these tutorials, you will describe, model, and solve a simple problem that also appears elsewhere in C, C++, and Java chapters of this manual:

- ◆ *Building and solving a small LP model in C*
- ◆ *Building and solving a small LP model in C++*
- ◆ *Building and solving a small LP model in Java*

Describe

The first step is for you to describe the problem in natural language and answer basic questions about the problem.

- ◆ What is the known information in this problem? That is, what data is available?
- ◆ What is the unknown information in this problem? That is, what are the decision variables?
- ◆ What are the limitations in the problem? That is, what are the constraints on the decision variables?
- ◆ What is the purpose of solving this problem? That is, what is the objective function?

Note: Though the **Describe** step of the process may seem trivial in a simple problem like this one, you will find that taking the time to fully describe a more complex problem is vital for creating a successful application. You will be able to code your application more quickly and effectively if you take the time to describe the model, isolating the decision variables, constraints, and objective.

Model

The second stage is for you to use the classes of ILOG Concert Technology for .NET users to build a model of the problem. The model is composed of *decision variables* and *constraints* on those variables. The model of this problem also contains an *objective*.

Solve

The third stage is for you to use an instance of the class `Cplex` to search for a solution and to solve the problem. Solving the problem consists of finding a value for each variable while simultaneously satisfying the constraints and minimizing the objective.

Describe

The aim in this tutorial is to see three different ways to build a model: by rows, by columns, or by nonzeros. After building the model of the problem in one of those ways, the application optimizes the problem and displays the solution.

Step One: Describe the problem

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

- ◆ What is known about the problem?
- ◆ What are the unknown pieces of information (the decision variables) in this problem?
- ◆ What are the limitations (the constraints) on the decision variables?
- ◆ What is the purpose (the objective) of solving this problem?

Building a small LP problem in C#

Here is a conventional formulation of the problem that the example optimizes:

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

- ◆ What are the decision variables in this problem?

$$x_1, x_2, x_3$$

- ◆ What are the constraints?

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

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

$$0 \leq x_1 \leq 40$$

$$0 \leq x_2 \leq 40$$

$$0 \leq x_3 \leq 40$$

◆ What is the objective?

$$\text{Maximize } x_1 + 2x_2 + 3x_3$$

Model

After you have written a description of the problem, you can use classes of ILOG Concert Technology for .NET users with ILOG CPLEX to build a model.

Step 2: Open the file

Open the file *yourCPLEXhome* \examples\src\tutorials\LPex1lesson.cs in your integrated development environment, such as Microsoft Visual Studio.

Step 3: Create the model object

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

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

That statement creates an empty instance of the class `Cplex`. In the next steps, you will add methods that make it possible for your application populate the model with data, either by rows, by columns, or by nonzeros.

Step 4: Populate the model by rows

Now go to the comment **Step 4** in that file, and add these lines to create a method to populate the empty model with data by rows.

```
internal static void PopulateByRow(IMPModeler model,
                                   INumVar[][] var,
                                   IRange[][] rng) {
    double[] lb = {0.0, 0.0, 0.0};
    double[] ub = {40.0,
                  System.Double.MaxValue,
                  System.Double.MaxValue};
    INumVar[] x = model.NumVarArray(3, lb, ub);
    var[0] = x;

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

    rng[0] = new IRange[2];
    rng[0][0] = model.AddLe(model.Sum(model.Prod(-1.0, x[0]),
                                       model.Prod( 1.0, x[1]),
                                       model.Prod( 1.0, x[2])), 20.0);
    rng[0][1] = model.AddLe(model.Sum(model.Prod( 1.0, x[0]),
```

```

        model.Prod(-3.0, x[1]),
        model.Prod( 1.0, x[2])), 30.0);
    }

```

Those lines populate the model with data specific to this particular example. However, you can see from its use of the interface `IMPModeler` how to add *ranged constraints* to a model. `IMPModeler` is the Concert Technology interface typically used to build math programming (MP) matrix models. You will see its use again in Step 5 and Step 6.

Step 5: Populate the model by columns

Go to the comment **Step 5** in the file, and add these lines to create a method to populate the empty model with data by columns.

```

internal static void PopulateByColumn(IMPModeler model,
                                     INumVar[][] var,
                                     IRange[][] rng) {
    IObjective obj = model.AddMaximize();

    rng[0] = new IRange[2];
    rng[0][0] = model.AddRange(-System.Double.MaxValue, 20.0);
    rng[0][1] = model.AddRange(-System.Double.MaxValue, 30.0);

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

    var[0] = new INumVar[3];
    var[0][0] = model.NumVar(model.Column(obj, 1.0).And(
        model.Column(r0, -1.0).And(
            model.Column(r1, 1.0))),
        0.0, 40.0);
    var[0][1] = model.NumVar(model.Column(obj, 2.0).And(
        model.Column(r0, 1.0).And(
            model.Column(r1, -3.0))),
        0.0, System.Double.MaxValue);
    var[0][2] = model.NumVar(model.Column(obj, 3.0).And(
        model.Column(r0, 1.0).And(
            model.Column(r1, 1.0))),
        0.0, System.Double.MaxValue);
}

```

Again, those lines populate the model with data specific to this problem. From them you can see how to use the interface `IMPModeler` to add *columns* to an empty model.

While for many examples population by rows may seem most straightforward and natural, there are some models where population by columns is a more natural or more efficient approach to implement. For example, problems with network structure typically lend themselves

well to modeling by column. Readers familiar with matrix algebra may view the method `populateByColumn` as the transpose of `populateByRow`.

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

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

An `IColumn` object created with the method `ICplex.Column` contains information about how to use this column to introduce a new variable into an existing modeling object. For example if `obj` is an `IObjective` object, `cplex.Column(obj, 2.0)` creates an `IColumn` object containing the information to install a new variable in the expression of the `IObjective` object `obj` with a linear coefficient of `2.0`. Similarly, for an `IRange` constraint `rng`, the method call `cplex.Column(rng, -1.0)` creates an `IColumn` object containing the information to install a new variable into the expression of `rng`, as a linear term with coefficient `-1.0`.

In short, when you use a modeling-by-column approach, new columns are created and installed as variables in all existing modeling objects where they are needed. To do this with ILOG Concert Technology, you create an `IColumn` object for every modeling object in which you want to install a new variable, and link them together with the method `IColumn.And`.

Step 6: Populate the model by nonzeros

Go to the comment **Step 6** in the file, and add these lines to create a method to populate the empty model with data by nonzeros.

```
internal static void PopulateByNonzero(IMPModeler model,
                                     INumVar[][] var,
                                     IRange[][] rng) {
    double[] lb = {0.0, 0.0, 0.0};
    double[] ub = {40.0, System.Double.MaxValue, System.Double.MaxValue};
;
    INumVar[] x = model.NumVarArray(3, lb, ub);
    var[0] = x;

    double[] objvals = {1.0, 2.0, 3.0};
    model.Add(model.Maximize(model.ScalProd(x, objvals)));

    rng[0] = new IRange[2];
    rng[0][0] = model.AddRange(-System.Double.MaxValue, 20.0);
    rng[0][1] = model.AddRange(-System.Double.MaxValue, 30.0);
}
```

```

    rng[0][0].Expr = model.Sum(model.Prod(-1.0, x[0]),
                               model.Prod( 1.0, x[1]),
                               model.Prod( 1.0, x[2]));
    rng[0][1].Expr = model.Sum(model.Prod( 1.0, x[0]),
                               model.Prod(-3.0, x[1]),
                               model.Prod( 1.0, x[2]));
}

```

In those lines, you can see how to populate an empty model with data indicating the nonzeros of the constraint matrix. Those lines first create objects for the objective and the ranges without expressions. They also create variables without columns; that is, variables with only their bounds. Then those lines create *expressions* over the objective, ranges, and variables and add the expressions to the model.

Step 7: Add an interface

Go to the comment **Step 7** in the file, and add these lines to create a method that tells a user how to invoke this application.

```

internal static void Usage() {
    System.Console.WriteLine("usage:  LPex1 <option>");
    System.Console.WriteLine("options: -r  build model row by row");
    System.Console.WriteLine("options: -c  build model column by column");
    System.Console.WriteLine("options: -n  build model nonzero by nonzero");
}

```

Step 8: Add a command evaluator

Go to the comment **Step 8** in the file, and add these lines to create a switch statement that evaluates the command that a user of your application might enter.

```

switch ( args[0].ToCharArray()[1] ) {
case 'r': PopulateByRow(cplex, var, rng);
         break;
case 'c': PopulateByColumn(cplex, var, rng);
         break;
case 'n': PopulateByNonzero(cplex, var, rng);
         break;
default: Usage();
         return;
}

```

Solve

After you have declared the decision variables and added the constraints and objective function to the model, your application is ready to search for a solution.

Step 9: Search for a solution

Go to **Step 9** in the file, and add this line to make your application search for a solution.

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

Step 10: Display the solution

Go to the comment **Step 10** in the file, and add these lines to enable your application to display any solution found in Step 9.

```
double[] x      = cplex.GetValues(var[0]);
double[] dj     = cplex.GetReducedCosts(var[0]);
double[] pi     = cplex.GetDuals(rng[0]);
double[] slack  = cplex.GetSlacks(rng[0]);

cplex.Output().WriteLine("Solution status = "
    + cplex.GetStatus());
cplex.Output().WriteLine("Solution value = "
    + cplex.ObjValue);

int nvars = x.Length;
for (int j = 0; j < nvars; ++j) {
    cplex.Output().WriteLine("Variable  :"
        + j
        + " Value = "
        + x[j]
        + " Reduced cost = "
        + dj[j]);
}

int ncons = slack.Length;
for (int i = 0; i < ncons; ++i) {
    cplex.Output().WriteLine("Constraint:"
        + i
        + " Slack = "
        + slack[i]
        + " Pi = "
        + pi[i]);
}
```

```
}
```

Step 11: Save the model to a file

If you want to save your model to a file in LP format, go to the comment **Step 11** in your application file, and add this line.

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

If you have followed the steps in this tutorial interactively, you now have a complete application that you can compile and execute.

Complete program

You can view the complete program online in the standard distribution of the product at `yourCPLEXinstallation\examples\src\LPex1.cs`.

Callable Library tutorial

Shows how to write applications that use the ILOG CPLEX Callable Library (C API).

In this section

The design of the ILOG CPLEX Callable Library

Describes the architecture of the Callable Library (C API).

Compiling and linking Callable Library applications

Documents compilation and linking an application of the C API.

How ILOG CPLEX works

Describes activities of ILOG CPLEX when it is invoked from the Callable Library (C API).

Creating a successful Callable Library application

Suggests guidelines for successful applications of the C API.

Building and solving a small LP model in C

Demonstrates an application in the C API.

Reading a problem from a file: example lpex2.c

Demonstrates an application that reads the model from a formatted file.

Adding rows to a problem: example lpex3.c

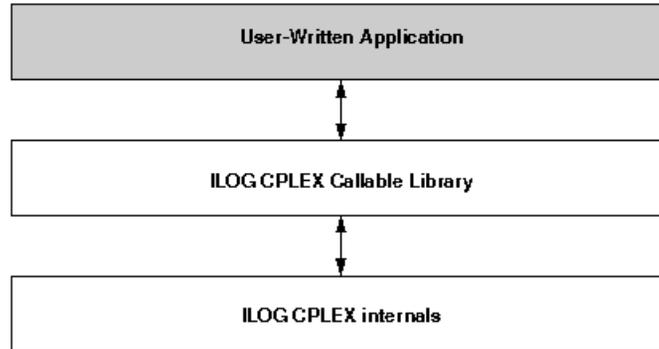
Demonstrates an application to add rows to a model.

Performing sensitivity analysis

Demonstrates sensitivity analysis in an application of the C API.

The design of the ILOG CPLEX Callable Library

A View of the ILOG CPLEX Callable Library shows a picture of the ILOG CPLEX world. The ILOG CPLEX Callable Library together with the ILOG CPLEX internals make up the ILOG CPLEX core. The core becomes associated with your application through Callable Library routines. The ILOG CPLEX environment and all problem-defining data are established inside the ILOG CPLEX core.



A View of the ILOG CPLEX Callable Library

The ILOG CPLEX Callable Library includes several categories of routines:

- ◆ optimization and result routines for defining a problem, optimizing it, and getting the results;
- ◆ utility routines for addressing application programming matters;
- ◆ problem modification routines to change a problem after it has been created within the ILOG CPLEX internals;
- ◆ problem query routines to access information about a problem after it has been created;
- ◆ file reading and writing routines to move information from the file system into your application or out of your application to the file system;
- ◆ parameter setting and query routines to access and modify the values of control parameters maintained by ILOG CPLEX.

Compiling and linking Callable Library applications

Documents compilation and linking an application of the C API.

In this section

Overview

Explains compilation and linking for Callable Library applications.

Building Callable Library applications on UNIX platforms

Describes special considerations for compiling and linking on UNIX platforms.

Building Callable Library applications on Win32 platforms

Describes special considerations about compiling and linking on various Windows platforms.

Overview

Each Callable Library is distributed as a single library file `libcplex.a` or `cplexXXX.lib`. Use of the library file is similar to that with `.o` or `.obj` files. Simply substitute the library file in the link procedure. This procedure simplifies linking and makes sure that the smallest possible executable is generated.

The following compilation and linking instructions assume that the example source programs and ILOG CPLEX Callable Library files are in the directories associated with a default installation of the software. If this is not true, additional compile and link flags would be required to point to the locations of the include file `cplex.h`, and Callable Library files respectively.

Note: The instructions below were current at the time of publication. As compilers, linkers and operating systems are released, different instructions may apply. Be sure to check the *Release Notes* that come with your ILOG CPLEX distribution for any changes. Also check the ILOG CPLEX web page (<http://www.ilog.com/products/cplex>).

Building Callable Library applications on UNIX platforms

To compile and execute an example (`lpex1`) do the following:

```
IloNumVar[] x = cplex.numVarArray(3, 0.0, 100.0);
x[0] + 2*x[1] + 3*x[2]

IloNumExpr expr = cplex.sum(x[0],
                            cplex.prod(2.0, x[1]),
                            cplex.prod(3.0, x[2]));
IloLinearNumExpr expr = cplex.linearNumExpr();
expr.addTerm(1.0, x[0]);
expr.addTerm(2.0, x[1]);
expr.addTerm(3.0, x[2]);
IloObjective obj = cplex.minimize(expr);
cplex.add(obj);
cplex.addMinimize(expr);
cplex.addLe(cplex.sum(cplex.negative(x[0]), x[1], x[2]), 20);
double objval = cplex.getObjValue();
double[] xval = cplex.getValues(x);
var[0][0]
= model.numVar(model.column(obj, 1.0).and(
                model.column(r0, -1.0).and(
                model.column(r1, 1.0))),
                0.0, 40.0);

% cd examples/platform/format
% make lpex1 # to compile and execute the first CPLEX example
```

In that command, *platform* indicates the name of the subdirectory corresponding to your type of machine, and *format* indicates your particular library format, such as static, multi-threaded, and so forth.

A list of all the examples that can be built this way is to be found in the makefile by looking for `C_EX` (C examples), or you can view the files listed in `examples/src`.

The makefile contains recommended compiler flags and other settings for your particular computer, which you can find by searching in it for "Compiler options" and use in your applications that call ILOG CPLEX.

Building Callable Library applications on Win32 platforms

Building an ILOG CPLEX application using Microsoft Visual C++ Integrated Development Environment, or the Microsoft Visual C++ command line compiler are explained here.

Microsoft Visual C++ IDE

To make an ILOG CPLEX Callable Library application using Visual C++, first create or open a project in the Visual C++ Integrated Development Environment (IDE). Project files are provided for each of the examples found in the directory or folder `examples\platform\format` where *platform* and *format* refer to your type of machine and compiler. For details about the build process, refer to the information file `msvc.html`, which is found in the top of the installed ILOG CPLEX directory structure.

Note: The distributed application must be able to locate `CPLEXXXX.dll` at run time.

Microsoft Visual C++ Command Line Compiler

If the Visual C++ command line compiler is used outside of the IDE, the command should resemble the following example. The example command assumes that the file `cplexXXX.lib` is in the current directory with the source file `lpex1.c`, and that the line in the source file `"#include <ilcplex/cplex.h>"` correctly points to the location of the include file or else has been modified to do so (or that the directories containing these files have been added to the environment variables `LIB` and `INCLUDE` respectively).

```
cl lpex1.c cplexXXX.lib
```

This command will create the executable file `lpex1.exe`.

Using Dynamic Loading

Some projects require more precise control over the loading and unloading of DLLs. For information on loading and unloading DLLs without using static linking, please refer to the compiler documentation or to a book such as *Advanced Windows* by Jeffrey Richter from Microsoft Press. If this is not a requirement, the static link implementations already mentioned are easier to use.

How ILOG CPLEX works

Describes activities of ILOG CPLEX when it is invoked from the Callable Library (C API).

In this section

Overview

Introduces basic steps in a Callable Library application.

Opening the ILOG CPLEX environment

Describes initialization of the environment.

Instantiating the problem object

Describes creation of a problem object.

Populating the problem object

Describes populating the problem object with data.

Changing the problem object

Describes special considerations about modification of the problem object.

Overview

When your application uses routines of the ILOG CPLEX Callable Library, it must first open the ILOG CPLEX environment, then create and populate a problem object before it solves a problem. Before it exits, the application must also free the problem object and release the ILOG CPLEX environment. The following sections explain those steps.

Opening the ILOG CPLEX environment

ILOG CPLEX requires a number of internal data structures in order to execute properly. These data structures must be initialized before any call to the ILOG CPLEX Callable Library. The first call to the ILOG CPLEX Callable Library is always to the function `CPXopenCPLEX`. This routine checks for a valid ILOG CPLEX license and returns a pointer to the ILOG CPLEX environment. This pointer is then passed to every ILOG CPLEX Callable Library routine, except those, such as `CPXmsg`, which do not require an environment.

The application developer must make an independent decision as to whether the variable containing the environment pointer is a global or local variable. Multiple environments are allowed, but extensive opening and closing of environments may create significant overhead on the licensor and degrade performance; typical applications make use of only one environment for the entire execution, since a single environment may hold as many problem objects as the user wishes. After all calls to the Callable Library are complete, the environment is released by the routine `CPXcloseCPLEX`. This routine indicates to ILOG CPLEX that all calls to the Callable Library are complete, any memory allocated by ILOG CPLEX is returned to the operating system, and the use of the ILOG CPLEX license is ended for this run.

Instantiating the problem object

A *problem object* is instantiated (created and initialized) by ILOG CPLEX when you call the routine `CPXcreateprob`. It is destroyed when you call `CPXfreeprob`. ILOG CPLEX allows you to create more than one problem object, although typical applications will use only one. Each problem object is referenced by a pointer returned by `CPXcreateprob` and represents one specific problem instance. Most Callable Library functions (except parameter setting functions and message handling functions) require a pointer to a problem object.

Note: An attempt to use a problem object in any environment other than the environment (or a child of that environment) where the problem object was created will raise an error.

Populating the problem object

The problem object instantiated by `CPXcreateprob` represents an empty problem that contains no data; it has zero constraints, zero variables, and an empty constraint matrix. This empty problem object must be populated with data. This step can be carried out in several ways.

- ◆ The problem object can be populated by assembling arrays of data and then calling `CPXcopylp` to copy the data into the problem object. (For example, see *Building and solving a small LP model in C*.)
- ◆ Alternatively, you can populate the problem object by sequences of calls to the routines `CPXnewcols`, `CPXnewrows`, `CPXaddcols`, `CPXaddrows`, and `CPXchgcoeflist`; these routines may be called in any order that is convenient. (For example, see *Adding rows to a problem: example lpex3.c*.)
- ◆ If the data already exist in a file using MPS format or LP format, you can use `CPXreadcopyprob` to read the file and copy the data into the problem object. (For example, see *Reading a problem from a file: example lpex2.c*.)

Changing the problem object

A major consideration in the design of ILOG CPLEX is the need to efficiently re-optimize modified linear programs. In order to accomplish that, ILOG CPLEX must be aware of changes that have been made to a linear program since it was last optimized. Problem modification routines are available in the Callable Library.

Do not change the problem by changing the original problem data arrays and then making a call to `CPXcopylp`. Instead, change the problem using the problem modification routines, allowing ILOG CPLEX to make use of as much solution information as possible from the solution of the problem before the modifications took place.

For example, suppose that a problem has been solved, and that the user has changed the upper bound on a variable through an appropriate call to the ILOG CPLEX Callable Library. A re-optimization would then begin from the previous optimal basis, and if that old basis were still optimal, then that information would be returned without even the need to refactor the old basis.

Creating a successful Callable Library application

Suggests guidelines for successful applications of the C API.

In this section

Overview

Outlines useful steps in developing a Callable Library application.

Prototype the model

Recommends creating a small model for prototyping.

Identify the routines to call

Recommends identifying Callable Library routines to use.

Test procedures in the application

Recommends testing in the Interactive Optimizer.

Assemble the data

Recommends ways to populate the model with data.

Choose an optimizer

Recommends identifying the optimizer to use, according to problem type.

Observe good programming practices

Recommends programming practices described in the User's Manual.

Debug your program

Recommends a debugger and tips described in the User's Manual.

Test your application

Recommends testing performance and correctness.

Use the examples

Recommends examples to follow.

Overview

Callable Library applications are created to solve a wide variety of problems. Each application shares certain common characteristics, regardless of its apparent uniqueness. The following steps can help you minimize development time and get maximum performance from your programs:

Prototype the model

Create a small model of the problem to be solved. An algebraic modeling language is sometimes helpful during this step.

Identify the routines to call

By separating the application into smaller parts, you can easily identify the tools needed to complete the application. Part of this process consists of identifying the Callable Library routines that will be called.

In some applications, the Callable Library is a small part of a larger program. In that case, the only ILOG CPLEX routines needed may be for:

- ◆ problem creation;
- ◆ optimizing;
- ◆ obtaining results.

In other cases the Callable Library is used extensively in the application. If so, Callable Library routines may also be needed to:

- ◆ modify the problem;
- ◆ set parameters;
- ◆ manage input and output messages and files;
- ◆ query problem data.

Test procedures in the application

It is often possible to test the procedures of an application in the ILOG CPLEX Interactive Optimizer with a small prototype of the model. Doing so will help identify the Callable Library routines required. The test may also uncover any flaws in procedure logic before you invest significant development effort.

Trying the ILOG CPLEX Interactive Optimizer is an easy way to decide the best optimization procedure and parameter settings.

Assemble the data

You must decide which approach to populating the problem object is best for your application. Reading an MPS or LP file may reduce the coding effort but can increase the run-time and disk-space requirements of the program. Building the problem in memory and then calling `CPXcopylp` avoids time consuming disk-file reading. Using the routines `CPXnewcols`, `CPXnewrows`, `CPXaddcols`, `CPXaddrows`, and `CPXchgcoeflist` can lead to modular code that may be more easily maintained than if you assemble all model data in one step.

Another consideration is that if the Callable Library application reads an MPS or LP formatted file, usually another application is required to generate that file. Particularly in the case of MPS files, the data structures used to generate the file could almost certainly be used to build the problem-defining arrays for `CPXcopylp` directly. The result would be less coding and a faster, more efficient application. These observations suggest that formatted files may be useful when prototyping your application, while assembling the arrays in memory may be a useful enhancement for a production application.

Choose an optimizer

After a problem object has been instantiated and populated, it can be solved using one of the optimizers provided by the ILOG CPLEX Callable Library. The choice of optimizer depends on the problem type.

- ◆ LP and QP problems can be solved by:
 - the primal simplex optimizer;
 - the dual simplex optimizer; and
 - the barrier optimizer.
- ◆ LP and QP problems with a substantial network can also be solved by a special network optimizer.
- ◆ LP problems can also be solved by:
 - the sifting optimizer; and
 - the concurrent optimizer.
- ◆ If the problem includes integer variables, mixed integer programming (MIP) must be used.

There are also many different possible parameter settings for each optimizer. The default values will usually be the best for linear programs. Integer programming problems are more sensitive to specific settings, so additional experimentation will often be useful.

Choosing the best way to solve the problem can dramatically improve performance. For more information, refer to the sections about tuning LP performance and trouble-shooting MIP performance in the *ILOG CPLEX User's Manual*.

Observe good programming practices

Using good programming practices will save development time and make the program easier to understand and modify. A list of good programming practices is provided in the *ILOG CPLEX User's Manual*, in *Developing CPLEX applications*.

Debug your program

Your program may not run properly the first time you build it. Learn to use a symbolic debugger and other widely available tools that support the creation of error-free code. Use the list of debugging tips provided in the *ILOG CPLEX User's Manual* to find and correct problems in your Callable Library application.

Test your application

After an application works correctly, it still may have errors or features that inhibit execution speed. To get the most out of your application, be sure to test its performance as well as its correctness. Again, the ILOG CPLEX Interactive Optimizer can help. Since the Interactive Optimizer uses the same routines as the Callable Library, it should take the same amount of time to solve a problem as a Callable Library application.

Use the `CPXwriteprob` routine with the SAV format to create a binary representation of the problem object, then read it in and solve it with the Interactive Optimizer. If the application sets optimization parameters, use the same settings with the Interactive Optimizer. If your application takes significantly longer than the Interactive Optimizer, performance within your application can probably be improved. In such a case, possible performance inhibitors include fragmentation of memory, unnecessary compiler and linker options, and coding approaches that slow the program without causing it to give incorrect results.

Use the examples

The ILOG CPLEX Callable Library is distributed with a variety of examples that illustrate the flexibility of the Callable Library. The C source of all examples is provided in the standard distribution. For explanations about the examples of quadratic programming problems (QPs), mixed integer programming problems (MIPs) and network flows, see the *ILOG CPLEX User's Manual*. Explanations of the following examples of LPs appear in this manual:

`lpex1.c` illustrates various ways of generating a problem object.

`lpex2.c` demonstrates how to read a problem from a file, optimize it via a choice of several means, and obtain the solution.

`lpex3.c` demonstrates how to add rows to a problem object and reoptimize.

It is a good idea to compile, link, and run all of the examples provided in the standard distribution.

Building and solving a small LP model in C

The example `lpex1.c` shows you how to use problem modification routines from the ILOG CPLEX Callable Library in three different ways to build a model. The application in the example takes a single command line argument that indicates whether to build the constraint matrix by rows, columns, or nonzeros. After building the problem, the application optimizes it and displays the solution. Here is the problem that the example optimizes:

```
Maximize       $x_1 + 2x_2 + 3x_3$ 

subject to     $-x_1 + x_2 + x_3 \leq 20$ 
               $x_1 - 3x_2 + x_3 \leq 30$ 

with these bounds   $0 \leq x_1 \leq 40$ 
                   $0 \leq x_2 \leq$ 
                   $0 \leq x_3 \leq$ 
```

Before any ILOG CPLEX Callable Library routine can be called, your application must call the routine `CPXopenCPLEX` to get a pointer (called `env`) to the ILOG CPLEX environment. Your application will then pass this pointer to every Callable Library routine. If this routine fails, it returns an error code. This error code can be translated to a string by the routine `CPXgeterrorstring`.

After the ILOG CPLEX environment is initialized, the ILOG CPLEX screen indicator parameter (`CPX_PARAM_SCRIND`) is turned on by the routine `CPXsetintparam`. This causes all default ILOG CPLEX output to appear on the screen. If this parameter is not set, then ILOG CPLEX will generate no viewable output on the screen or in a file.

At this point, the routine `CPXcreateprob` is called to create an empty problem object. Based on the problem-building method selected by the command-line argument, the application then calls a routine to build the matrix by rows, by columns, or by nonzeros. The routine `populatebyrow` first calls `CPXnewcols` to specify the column-based problem data, such as the objective, bounds, and variables names. The routine `CPXaddrows` is then called to supply the constraints. The routine `populatebycolumn` first calls `CPXnewrows` to specify the row-based problem data, such as the righthand side values and sense of constraints. The routine `CPXaddcols` is then called to supply the columns of the matrix and the associated column bounds, names, and objective coefficients. The routine `populatebynonzero` calls both `CPXnewrows` and `CPXnewcols` to supply all the problem data except the actual constraint matrix. At this point, the rows and columns are well defined, but the constraint matrix remains empty. The routine `CPXchgcoeflist` is then called to fill in the nonzero entries in the matrix.

After the problem has been specified, the application optimizes it by calling the routine `CPXlpopt`. Its default behavior is to use the ILOG CPLEX Dual Simplex Optimizer. If this routine returns a nonzero result, then an error occurred. If no error occurred, the application allocates arrays for solution values of the primal variables, dual variables, slack variables, and reduced costs; then it obtains the solution information by calling the routine `CPXsolution`. This routine returns the status of the problem (whether optimal, infeasible, or unbounded, and whether a time limit or iteration limit was reached), the objective value and the solution vectors. The application then displays this information on the screen.

As a debugging aid, the application writes the problem to a ILOG CPLEX LP file (named `lpex1.lp`) by calling the routine `CPXwriteprob`. This file can be examined to detect whether any errors occurred in the routines creating the problem. `CPXwriteprob` can be called at any time after `CPXcreateprob` has created the `lp` pointer.

The label `TERMINATE` is used as a place for the program to exit if any type of failure occurs, or if everything succeeds. In either case, the problem object represented by `lp` is released by the call to `CPXfreeprob`, and any memory allocated for solution arrays is freed. The application then calls `CPXcloseCPLEX`; it tells ILOG CPLEX that all calls to the Callable Library are complete. If an error occurs when this routine is called, then a call to `CPXgeterrorstring` is needed to retrieve the error message, since `CPXcloseCPLEX` causes no screen output.

You can view the complete program online in the standard distribution of the product at *yourCPLEXinstallation* / `examples/src/lpex1.c`.

Reading a problem from a file: example lpex2.c

The previous example, `lpex1.c`, shows a way to copy problem data into a problem object as part of an application that calls routines from the ILOG CPLEX Callable Library. Frequently, however, a file already exists containing a linear programming problem in the industry standard MPS format, the ILOG CPLEX LP format, or the ILOG CPLEX binary SAV format. In example `lpex2.c`, ILOG CPLEX file-reading and optimization routines read such a file to solve the problem.

Example `lpex2.c` uses command line arguments to specify the name of the input file and the optimizer to call.

Usage: `lpex2 filename optimizer`

Where: `filename` is a file with extension MPS, SAV, or LP (lower case is allowed), and `optimizer` is one of the following letters:

- o default
- p primal simplex
- d dual simplex
- n network with dual simplex cleanup
- h barrier with crossover
- b barrier without crossover
- s sifting
- c concurrent

For example, this command:

```
lpex2 example.mps d
```

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

To illustrate the ease of reading a problem, the example uses the routine `CPXreadcopyprob`. This routine detects the type of the file, reads the file, and copies the data into the ILOG CPLEX problem object that is created with a call to `CPXcreateprob`. The user need not be concerned with the memory management of the data. Memory management is handled transparently by `CPXreadcopyprob`.

After calling `CPXopenCPLEX` and turning on the screen indicator by setting the `CPX_PARAM_SCRIND` parameter to `CPX_ON`, the example creates an empty problem object with a call to `CPXcreateprob`. This call returns a pointer, `lp`, to the new problem object. Then the data is read in by the routine `CPXreadcopyprob`. After the data is copied, the appropriate optimization routine is called, based on the command line argument.

After optimization, a call to `CPXgetstat` retrieves the status of the solution. The cases of infeasibility or unboundedness in the model are handled in a simple fashion here; a more complex application program might treat these cases in more detail. With these two cases out of the way, the program then calls `CPXsolninfo` to examine the nature of the solution. Certain that a solution exists, the application then calls `CPXgetobjval` to obtain the objective function value for this solution and report it.

Next, preparations are made to print the solution value and basis status of each individual variable, by allocating arrays of appropriate size; these sizes are detected by calls to the routines `CPXgetnumcols` and `CPXgetnumrows`. Note that a basis is not guaranteed to exist, depending on which optimizer was selected at run time, so some of these steps, including the call to `CPXgetbase`, are dependent on the solution type returned by `CPXsolninfo`.

The primal solution values of the variables are obtained by a call to `CPXgetx`, and then these values (along with the basis statuses if available) are printed, in a loop, for each variable. After that, a call to `CPXgetdblquality` provides a measure of the numerical roundoff error present in the solution, by obtaining the maximum amount by which any variable's lower or upper bound is violated.

After the `TERMINATE:` label, the data for the solution (`x`, `cstat`, and `rstat`) are freed. Then the problem object is freed by `CPXfreeprob`. After the problem is freed, the ILOG CPLEX environment is freed by `CPXcloseCPLEX`.

You can view the complete program online in the standard distribution of the product at *yourCPLEXinstallation/examples/src/lpex2.c*.

Adding rows to a problem: example lpex3.c

This example illustrates how to develop your own solution algorithms with routines from the Callable Library. It also shows you how to add rows to a problem object. Here is the problem that lpex3 solves:

$$\begin{aligned} &\text{Minimize} && c^T x \\ &\text{subject to} && Hx = d \\ & && Ax = b \\ & && l \leq x \leq u \end{aligned}$$

where

$$\begin{aligned} H = & \begin{pmatrix} -1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & -1 \end{pmatrix} & d = \begin{pmatrix} -3 \\ 1 \\ 4 \\ 3 \\ -5 \end{pmatrix} \\ A = & \begin{pmatrix} 2 & 1 & -2 & -1 & 2 & -1 & -2 & -3 \\ 1 & -3 & 2 & 3 & -1 & 2 & 1 & 1 \end{pmatrix} & b = \begin{pmatrix} 4 \\ -2 \end{pmatrix} \\ c = & (-9 \ 1 \ 4 \ 2 \ -8 \ 2 \ 8 \ 12) \\ l = & (0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0) \\ u = & (50 \ 50 \ 50 \ 50 \ 50 \ 50 \ 50 \ 50) \end{aligned}$$

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

1. The ILOG CPLEX Network Optimizer is used to solve

$$\begin{aligned} &\text{Minimize} && c^T x \\ &\text{subject to} && Hx = d \end{aligned}$$

2. The constraints $Ax=b$ are added to the problem, and the dual simplex optimizer is used to solve the new problem, starting at the optimal basis of the simpler network problem.

The data for this problem consists of the network portion (using variable names beginning with the letter H) and the complicating constraints (using variable names beginning with the letter A).

The example first calls `CPXopenCPLEX` to create the environment and then turns on the ILOG CPLEX screen indicator (`CPX_PARAM_SCRIND`). Next it sets the simplex display level (`CPX_PARAM_SIMDISPLAY`) to 2 to indicate iteration-by-iteration output, so that the progress of each iteration of the optimizer can be observed. Setting this parameter to 2 is not generally recommended; the example does so only for illustrative purposes.

The example creates a problem object by a call to `CPXcreateprob`. Then the network data is copied via a call to `CPXcopylp`. After the network data is copied, the parameter `CPX_PARAM_LPMETHOD` is set to `CPX_ALG_NET` and the routine `CPXlpopt` is called to solve the network part of the optimization problem using the network optimizer. The objective value of this problem is retrieved by `CPXgetobjval`.

Then the extra rows are added by `CPXaddrows`. For convenience, the total number of nonzeros in the rows being added is stored in an extra element of the array `rmatbeg`, and this element is passed for the parameter `nzcnt`. The name arguments to `CPXaddrows` are `NULL`, since no variable or constraint names were defined for this problem.

After the `CPXaddrows` call, the parameter `CPX_PARAM_LPMETHOD` is set to `CPX_ALG_DUAL` and the routine `CPXlpopt` is called to re-optimize the problem using the dual simplex optimizer. After re-optimization, `CPXsolution` accesses the solution status, the objective value, and the primal solution. `NULL` is passed for the other solution values, since the information they provide is not needed in this example.

At the end, the problem is written as a SAV file by `CPXwriteprob`. This file can then be read into the ILOG CPLEX Interactive Optimizer to analyze whether the problem was correctly generated. Using a SAV file is recommended over MPS and LP files, as SAV files preserve the full numeric precision of the problem.

After the `TERMINATE: label`, `CPXfreeprob` releases the problem object, and `CPXcloseCPLEX` releases the ILOG CPLEX environment.

You can view the complete program online in the standard distribution of the product at *yourCPLEXinstallation/examples/src/lpex3.c*.

Performing sensitivity analysis

In *Performing sensitivity analysis*, there is a discussion of how to perform sensitivity analysis in the Interactive Optimizer. As with most interactive features of ILOG CPLEX, there is a direct approach to this task from the Callable Library. This section modifies the example `lpex1.c` in *Building and solving a small LP model in C* to show how to perform sensitivity analysis with routines from the Callable Library.

To begin, make a copy of `lpex1.c`, and edit this new source file. Among the declarations (for example, immediately after the declaration for `dj`) insert these additional declarations:

```
double *lowerc = NULL, *upperc = NULL;
double *lowerr = NULL, *upperr = NULL;
```

At some point after the call to `CPXlpopt`, (for example, just before the call to `CPXwriteprob`), perform sensitivity analysis on the objective function and on the righthand side coefficients by inserting this fragment of code:

```
upperc = (double *) malloc (cur_numcols * sizeof(double));
lowerc = (double *) malloc (cur_numcols * sizeof(double));
status = CPXobjsa (env, lp, 0, cur_numcols-1, lowerc, upperc);
if ( status ) {
    fprintf (stderr, "Failed to obtain objective sensitivity.\n");
    goto TERMINATE;
}
printf ("\nObjective coefficient sensitivity:\n");
for (j = 0; j < cur_numcols; j++) {
    printf ("Column %d: Lower = %10g Upper = %10g\n",
           j, lowerc[j], upperc[j]);
}

upperr = (double *) malloc (cur_numrows * sizeof(double));
lowerr = (double *) malloc (cur_numrows * sizeof(double));
status = CPXrhssa (env, lp, 0, cur_numrows-1, lowerr, upperr);
if ( status ) {
    fprintf (stderr, "Failed to obtain RHS sensitivity.\n");
    goto TERMINATE;
}
printf ("\nRight-hand side coefficient sensitivity:\n");
for (i = 0; i < cur_numrows; i++) {
    printf ("Row %d: Lower = %10g Upper = %10g\n",
           i, lowerr[i], upperr[i]);
}
```

This sample is familiarly known as “throw away” code. For production purposes, you probably want to observe good programming practices such as freeing these allocated arrays at the `TERMINATE` label in the application.

A bound value of $1e+20$ (`CPX_INFBOUND`) is treated as infinity within ILOG CPLEX, so this is the value printed by our sample code in cases where the upper or lower sensitivity range on a row or column is infinite; a more sophisticated program might print a string, such as `-inf` or `+inf`, when negative or positive `CPX_INFBOUND` is encountered as a value.

Similar code could be added to perform sensitivity analysis with respect to bounds via `CPXboundsa`.

ILOG CPLEX User's Manual

Welcome to ILOG CPLEX! Here's its user's manual, a practical guide to its features.

ILOG CPLEX User's Manual

Welcome to ILOG CPLEX! Here's its user's manual, a practical guide to its features.

In this section

Meet ILOG CPLEX

Introduces ILOG CPLEX, explains what it does, suggests prerequisites, and offers advice for using this documentation with it.

Languages and APIs

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

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:

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.

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.

Infeasibility and unboundedness

Documents tools to help you analyze the source of the infeasibility in a model: the preprocessing reduction parameter for distinguishing infeasibility from unboundedness, the conflict refiner for detecting minimal sets of mutually contradictory bounds and constraints, and FeasOpt for repairing infeasibilities.

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.

Meet ILOG CPLEX

Introduces ILOG CPLEX, explains what it does, suggests prerequisites, and offers advice for using this documentation with it.

In this section

What is ILOG CPLEX?

Describes ILOG CPLEX Component Libraries.

What does ILOG CPLEX do?

Defines the scope of ILOG CPLEX.

What you need to know

Suggests prerequisites for using ILOG CPLEX.

Examples online

Describes examples delivered with the product.

Notation in this manual

Documents notation in this manual.

Related documentation

Describes other available documentation of the product.

Announcements and updates

Describes mailing list.

Further reading

Recommends further reading about related topics.

What is 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.

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 `ilocplex112.lib`, `concert.lib` and `cplex112.jar` as well as `cplex112.dll` and `concert27.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 to refer 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:

$$\begin{array}{ll}
 \text{Minimize (or maximize)} & c_1 x_1 + c_2 x_2 + \dots + c_n x_n \\
 \\
 \text{subject to} & a_{11} x_1 + a_{12} x_2 + \dots + a_{1n} x_n \sim b_1 \\
 & a_{21} x_1 + a_{22} x_2 + \dots + a_{2n} x_n \sim b_2 \\
 & \dots \\
 & a_{m1} x_1 + a_{m2} x_2 + \dots + a_{mn} x_n \sim b_m \\
 \\
 \text{with these bounds} & l_1 \leq x_1 \leq u_1, \dots, l_n \leq x_n \leq u_n
 \end{array}$$

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:

$$\begin{array}{ll}
 \text{objective function coefficients} & c_1, \dots, c_n \\
 \\
 \text{constraint coefficients} & a_{11}, \dots, a_{mn} \\
 \\
 \text{righthand side} & b_1, \dots, b_m \\
 \\
 \text{upper bounds} & u_1, \dots, u_n \\
 \\
 \text{lower bounds} & l_1, \dots, l_n \\
 \\
 \text{variables or unknowns} & x_1, \dots, x_n
 \end{array}$$

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*.

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)*, 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)*, 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)*.

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* offers more detail about how the ILOG CPLEX Network Optimizer works.

What you need to know

The manual assumes that you are familiar with ILOG CPLEX from reading *Getting Started with ILOG CPLEX* and from following the tutorials there. 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* 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.

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. *Examples* lists the examples in this manual and indicates where to find them.

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>
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>
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>
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>
linear programming: starting from an advanced basis	<code>ilolpex6.cpp</code> <code>lpex6.c</code>	<i>Example ilolpex6.cpp</i> <i>Example lpex6.c</i>
network optimization: using the Callable Library	<code>netex1.c</code>	<i>Example: using the network optimizer with the Callable Library netex1.c</i>
network optimization: relaxing a network flow to an LP	<code>netex2.c</code>	<i>Example: network to LP transformation netex2.c</i>
quadratic programming: maximizing a QP	<code>iloqpex1.cpp</code> <code>QPex1.java</code> <code>qpex1.c</code>	<i>Example: iloqpex1.cpp</i> <i>Example: QPex1.java</i> <i>Example: qpex1.c</i>
quadratic programming: reading a QP from a formatted file	<code>qpex2.c</code>	<i>Example: reading a QP from a file qpex2.c</i>

Example	Source File	In This Manual
quadratically constrained programming: QCP	qcpex1.c iloqcpex1.cpp QCPex1.java	<i>Examples: QCP</i>
mixed integer programming: optimizing a basic MIP	ilomipex1.cpp mipex1.c	<i>Examples: optimizing a simple MIP problem</i>
mixed integer programming: reading a MIP from a formatted file	ilomipex2.cpp mipex2.c	<i>Example: reading a MIP problem from a file</i>
mixed integer programming: using special ordered sets (SOS) and priority orders	ilomipex3.cpp mipex3.c	<i>Example: using SOS and priority</i>
cutting stock: using column generation	cutstock.cpp	<i>What is column generation?</i>
transport: piecewise-linear optimization	transport.cpp	<i>Complete program: transport.cpp</i>
food manufacturing 2: using logical constraints	foodmanufac.cpp	<i>Using logical constraints: Food Manufacture 2</i>
early tardy scheduling	etsp.cpp	<i>Early tardy scheduling</i>
input and output: using the message handler	lpex5.c	<i>Example: Callable Library message channels</i>
using query routines	lpex7.c	<i>Example: using query routines lpex7.c</i>
using callbacks	ilolpex4.cpp lpex4.c iloadmipex5.cpp	<i>Example: deriving the simplex callback ilolpex4.cpp</i> <i>Example: using callbacks lpex4.c</i> <i>Example: controlling cuts iloadmipex5.cpp</i>
using the tuning tool	tuneset.c	<i>Meet the tuning tool and Example: time limits on tuning in the Interactive Optimizer</i>
mixed integer programming: solution pool	location.lp	<i>Example: simple facility location problem</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* documents 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: A^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 > CPLEX or in C:\ILOG\Optim\CPLEX .

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:

- tuning performance,
- managing input and output,
- generating and keeping multiple solutions in the solution pool,
- 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.

- ◆ 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 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:

- 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 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 CPLEX .NET Reference Manual: This manual documents the .NET API of Concert Technology for ILOG CPLEX. 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 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 format, available from your customer support website, or separately on your installation CD.

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**, fourth edition. 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.

Languages and APIs

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

In this section

ILOG Concert Technology for C++ users

Explores the features ILOG CPLEX offers to users of C++ to solve mathematical programming problems.

ILOG Concert Technology for Java users

Explores the features ILOG CPLEX offers to Java users to solve mathematical programming problems.

ILOG Concert Technology for .NET users

Explores the features that ILOG CPLEX offers to users of C#.NET through Concert Technology.

ILOG CPLEX Callable Library

Shows how to write C applications using the ILOG CPLEX Callable Library.

ILOG Concert Technology for C++ users

Explores the features ILOG CPLEX offers to users of C++ to solve mathematical programming problems.

In this section

Overview

Highlights the design, architecture, modeling facilities for C++ users of ILOG CPLEX.

Architecture of a CPLEX C++ application

Describes the architecture of a conventional ILOG CPLEX application in C++.

Licenses

Describes licensing conventions for a C++ application using ILOG CPLEX.

Compiling and linking

Tells where to find instructions to compile and link a C++ application using ILOG CPLEX.

Creating a C++ application with Concert Technology

Outlines steps to create a C++ application with ILOG CPLEX.

Modeling an optimization problem with Concert Technology

Introduces classes of the C++ API of Concert Technology for modeling optimization problems to be solved by IloCplex.

Solving the model

Describes facilities for solving a model in the C++ API.

Accessing solution information

Describes available information about solution feasibility, solution variables, basis information, and solution quality.

Modifying a model

Describes methods in the C++ API to modify a model.

Handling errors

Describes error handling in the C++ API.

Example: optimizing the diet problem in C++

Shows an example of the C++ API.

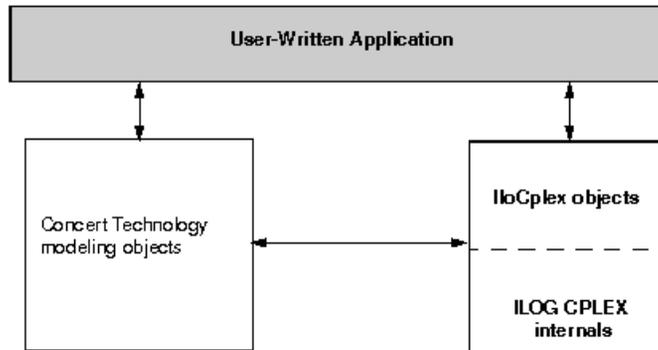
Overview

This topic explains the design of the library, explains modeling techniques, and offers an example of programming with Concert Technology in C++. It also provides information about controlling parameters in a C++ application. It shows how to write C++ programs using ILOG CPLEX Concert Technology for C++ users. It also includes information about licensing, compiling, and linking your programs.

Architecture of a CPLEX C++ application

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:

- ◆ 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.
- ◆ 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.



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

These are the steps most applications are likely to entail.

1. First, create a model of your problem with the modeling facilities of Concert Technology. *Modeling an optimization problem with Concert Technology* offers an introduction to creating a model.
2. When the model is ready to be solved, hand it over to ILOG CPLEX for solving. *Solving the model* 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.
3. After ILOG CPLEX solves the model, *Accessing solution information*, shows you how to access and interpret results from the optimization.
4. After analyzing the results, you may want to make changes to the model and study their effect. *Modifying a model* explains how to make changes and how ILOG CPLEX deals with them.
5. *Handling errors*, discusses the error handling and debugging support provided by Concert Technology and ILOG CPLEX.
6. *Example: optimizing the diet problem in C++* 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*.

Modeling an optimization problem with Concert Technology

Introduces classes of the C++ API of Concert Technology for modeling optimization problems to be solved by IloCplex.

In this section

Overview

Highlights the C++ classes for models in ILOG CPLEX.

Creating the environment: IloEnv

Describes the C++ class IloEnv.

Defining variables and expressions: IloNumVar

Describes the C++ class IloNumVar.

Declaring the objective: IloObjective

Describes the C++ class IloObjective.

Adding constraints: IloConstraint and IloRange

Describes the C++ classes for constraints and ranges.

Formulating a problem: IloModel

Describes the C++ class IloModel.

Managing data

Describes C++ classes to manage data.

Overview

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 *Concert Technology modeling objects in C++*.

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. Failure 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 to set 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*.

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 topics:

- ◆ *Solving LPs: simplex optimizers* and *Solving LPs: barrier optimizer* both discuss linear expressions.
- ◆ *Solving problems with a quadratic objective (QP)* discusses quadratic expressions in an objective function.
- ◆ *Solving problems with quadratic constraints (QCP)* 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* introduces piecewise-linear expressions through a transportation example.
- ◆ *Logical constraints in optimization* introduces logical constraints handled by ILOG CPLEX. Chapters following it offer examples.

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

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

Declaring the objective: IloObjective

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

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

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

Adding constraints: IloConstraint and IloRange

Similarly, objects of the class `IloConstraint` represents constraints in your model. Most constraints will belong to the subclass `IloRange`, derived from `IloConstraint`, and thus inherit its constructors and methods. `IloRange` represent constraints of the form $\text{lower bound} \leq \text{expression} \leq \text{upper bound}$. 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.

Managing data

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 the 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: optimizing the diet problem in C++* 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

Describes facilities for solving a model in the C++ API.

In this section

Overview

Introduces the C++ class IloCplex.

Extracting a model

Describes the method that extracts a model for an algorithm in the C++ API.

Invoking a solver

Describes the method that invokes a solver in the C++ API.

Choosing an optimizer

Describes the optimizers available in the C++ API.

Controlling the optimizers

Describes parameters that control the optimizers in the C++ API.

Overview

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 *Concert Technology modeling objects in C++*.

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)* and *Solving problems with quadratic constraints (QCP)*. Likewise, *Using piecewise linear functions in optimization: a transport example* 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.

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 <code><=</code> expression <code><=</code> upper bound) or

To model:	Use:
	objects of the class <code>IloConstraint</code> (<code>expr1 relation 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)*.

For more information about quadratic objective functions, see *Solving problems with a quadratic objective (QP)*.

For examples of piecewise linear constraints, see *Using piecewise linear functions in optimization: a transport example*.

For more about logical constraints, see *Logical constraints in optimization*.

For a description of special ordered sets, see *Using special ordered sets (SOS)*.

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(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(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 `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* and *Solving LPs: barrier optimizer* discuss the algorithms available for solving LPs. Similarly, *Solving problems with a quadratic objective (QP)*, discusses the algorithms available for solving QPs. *Solving problems with quadratic constraints (QCP)* re-introduces the barrier optimizer in the context of quadratically constrained programming problems (QCPs). *Using piecewise linear functions in optimization: a transport example* introduces piecewise-linear functions through a transportation example. *Logical constraints in optimization* introduces logical constraints, and chapters following it offer examples.
- ◆ In all other cases, the extracted problem that ILOG CPLEX sees is indeed a MIP and, in general, a *series* of continuous relaxations needs to be solved. The method `cplex.isMIP` returns `IloTrue` in such a case. *Solving mixed integer programming problems (MIP)* discusses the algorithms applied.

The optimizer option used for solving the first continuous relaxation (whether it is the only one or just the first in a series of problems) is controlled by 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. The table *Optimizer options in IloCplex::Algorithm* displays the meaning of the optimizer options defined by `IloCplex::Algorithm`.

Tip: The choice `Sifting` is not available for QP models. Only the `Barrier` option is available for QCP models. The table *Algorithm available at root by problem type* summarizes these options.

Optimizer options in IloCplex::Algorithm

`AutoAlg` let CPLEX decide which algorithm to use

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

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)
```

Algorithm types for NodeAlg summarizes the options available at nodes.

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. *Nested enumerations for integer parameters* summarizes those parameters and their enumeration types.

Nested enumerations for integer parameters

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:

- ◆ `setParam(BoolParam, value);`

- ◆ `setParam(IntParam, value);`
- ◆ `setParam(NumParam, value);`
- ◆ `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 this call:

```
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`s 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 you are solving a MIP, ILOG CPLEX provides additional controls of the solution process. Priority orders and branching directions can be used to control the branching in a static way. These controls are discussed in *Heuristics*. 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 you solve the model.

Dynamic control of the solution process of MIPs is provided through goals or control callbacks. They are discussed in *Using goals*, and in *Using optimization callbacks*. 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* contrasts the advantages of each approach.

Accessing solution information

Describes available information about solution feasibility, solution variables, basis information, and solution quality.

In this section

Accessing solution status

Describes the status of a solution.

Querying solution data

Describes methods available in the C++ API to query data about the solution after optimization.

Accessing basis information

Describes methods in the C++ API to retrieve basis information.

Performing sensitivity analysis

Describes methods in the C++ API to analyze infeasible problems.

Analyzing infeasible problems

Describes methods in the C++ API to analyze infeasible models.

Assessing solution quality

Describes methods in the C++ API to assess quality of a solution.

Accessing solution status

Calling `solve` returns a Boolean value 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, call the method `getStatus`. It returns a member of the nested enumeration `IloAlgorithm_Status`. The fully qualified names of those symbols have the `IloAlgorithm` prefix. *Algorithm status and information about the model* shows what each return status means for the extracted model.

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 you see, these statuses indicate information about the model that the ILOG CPLEX optimizer was able to prove during the most recent call to the 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 the method `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 `getValue` 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 = 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 array of numeric variables (`IloNumVarArray`) named `var`, use lines like these to access the solution values for all variables in `var` simultaneously:

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

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 `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 you solve 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 `getStatuses` 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 the methods `getBoundSA`, `getRHSSA`, and `getObjSA`, respectively.

Analyzing infeasible problems

An important feature of ILOG CPLEX is that even if no feasible solution has been found, (that is, if `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 `getCplexStatus` may help you decide 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 investigating 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*.

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*.

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

Assessing 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 the method `getQuality`, like this:

```
IloNum violation = 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 the method with these arguments instead:

```
IloRange maxrange; IloNumVar maxvar; IloNum violation =  
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

Describes methods in the C++ API to modify a model.

In this section

Overview

Outlines different ways to modify a model for ILOG CPLEX in a C++ application.

Deleting and removing modeling objects

Describes the effects of deleting modeling objects in the C++ API.

Changing variable type

Describes IloConversion, the class of the C++ API to change the type of a variable.

Overview

In some applications, you may want to solve another model or a modification of your original 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:

```
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:

```
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 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:

- ◆ 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.

- ◆ 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. You query the error code from a caught exception by calling the method:

```
IloInt getStatus () const;
```

You query the error message by calling the method:

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

That method 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 `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 this:

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

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

Example: optimizing the diet problem in C++

Shows an example of the C++ API.

In this section

Overview

Outlines how to solve the diet problem with ILOG CPLEX in a C++ application.

Problem representation

Describes how the problem is represented in the application.

Application description

Describes the architecture of the application.

Creating multi-dimensional arrays with IloArray

Describes data handling in the application.

Using arrays for input or output

Describes input and out in the application.

Solving the model with IloCplex

Shows how to solve the example.

Complete program

Refers to the online sample.

Overview

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* describes the problem.

The example `ilodiet.cpp` illustrates these procedures:

- ◆ *Creating a model row by row;*
- ◆ *Creating a model column by column.*

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

Notes about the application are available in *Complete program*.

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 decide 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 `...` is left to be filled in after 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 or output

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

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

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

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

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

Building the model by row

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

The statement:

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

creates the objective function and adds it to the model. The `IloScalProd` function creates the expression $\sum_j (Buy[j] * foodCost[j])$ which is then passed to the function `IloMinimize`. That function creates and returns the actual `IloObjective` object, which is added to the model with the call `mod.add`.

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

the overloaded operator `<=` to construct a range constraint (an `IloRange` object) which is added to the model:

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

After an expression has been used for creating a constraint, it is deleted by 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(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*.

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

ILOG Concert Technology for Java users

Explores the features ILOG CPLEX offers to Java users to solve mathematical programming problems.

In this section

Architecture of a CPLEX Java application

Describes the architecture of a Java application in Concert Technology.

Creating a Java application with Concert Technology

Introduces the components of an application in the Java API of Concert Technology and covers the steps most Java applications are likely to follow.

Modeling an optimization problem with Concert Technology in the Java API

Describes classes in the Java API to model an optimization problem.

Solving the model

Describes the class `IloCplex` in the Java API.

Accessing solution information

Describes the information available in the Java API about a solution.

Choosing an optimizer

Describes algorithms implemented in the Java API for solving optimization problems.

Controlling ILOG CPLEX optimizers

Describes the features that control optimizers in the Java API.

More solution information

Describes additional information available from a solution.

Advanced modeling with IloLPMatrix

Explains the concept of matrix modeling as implemented by the class IloLPMatrix in the Java API.

Modeling by column

Introduces modeling by columns as implemented in the Java API.

Example: optimizing the diet problem in Java

Describes an application to solve the diet problem in the Java API.

Modifying the model

Describes modification of a model in the Java API.

Architecture of a CPLEX Java application

Describes the architecture of a Java application in Concert Technology.

In this section

Overview

Offers an overview of the architecture.

Licenses in a Java application

Describes licensing conventions specific to the Java API.

Compiling and linking a Java application

Tells where to find the compilation and linking instructions specific to the Java API.

Overview

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.

A view of Concert Technology for Java users 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.

A user-written Java application and ILOG CPLEX internals use separate memory heaps. Java supports two different command-line options of interest in this respect:

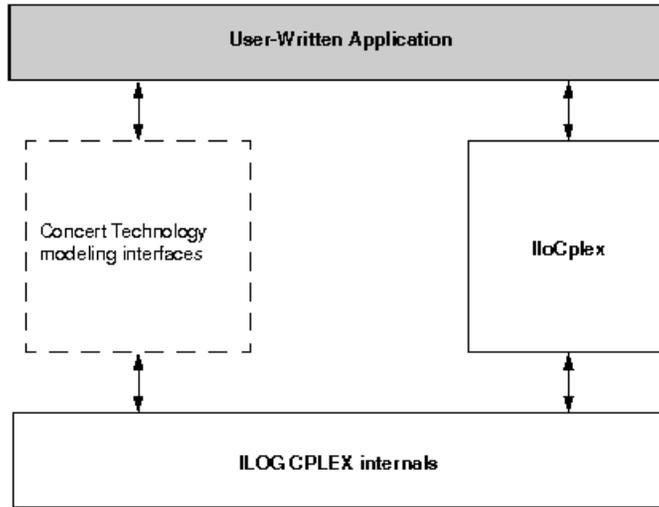
- ◆ `-Xms` sets the initial heap size for the Java-part of the application;
- ◆ `-Xmx` sets the maximum heap size for the Java part of the application.

For applications where more memory is needed, you must carefully assess whether the Java application needs to increase or decrease its memory heap. In some cases, the Java application needs more memory to create the model to be solved. If so, use the `-Xmx` option to increase the maximum heap size. Keep in mind that allocating more space in the heap to the Java application makes less memory available to the ILOG CPLEX internals, so increase the maximum heap size only as needed. Otherwise, the Java application will use too much memory, imposing an unnecessary limitation on the memory available to the ILOG CPLEX internals.

In cases where insufficient memory is available for ILOG CPLEX internals, there are remedies to consider with respect to the heap:

- ◆ make sure the maximum heap size has not been set to a value larger than needed;
- ◆ consider reducing the default maximum heap size if the Java application can operate with less memory.

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.



A view of Concert Technology for Java users

Licenses in a Java application

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 a Java application

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

First, create a model of your problem with the modeling facilities of Concert Technology. *Modeling an optimization problem with Concert Technology in the Java API* offers an introduction to creating a model. *Building the model* goes into more detail.

When the model is ready to be solved, hand it over to ILOG CPLEX for solving. *Solving the model* 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 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* 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 presents a complete program.

Not covered in this chapter are advanced features, such as the use of goals or callbacks to query data about an ongoing optimization and for controlling the optimization itself. Goals, callbacks, and other advanced features are discussed in *Advanced programming techniques*.

Modeling an optimization problem with Concert Technology in the Java API

Describes classes in the Java API to model an optimization problem.

In this section

Overview

Introduces classes to support models for ILOG CPLEX in Java applications.

Using IloModeler

Describes the class IloModeler in the Java API.

The active model

Describes the active model in the Java API.

Building the model

Uses the diet example to demonstrate building a model for ILOG CPLEX in Java.

Overview

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`.

Note: 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:

```
lowerbound ≤ expression ≤ upperbound
```

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

The `IloMIPModeler` 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.

Modeling classes of ILOG CPLEX with Concert Technology for Java users summarizes those observations about the interfaces of ILOG CPLEX with Concert Technology for Java users.

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>

To Model This	Use an Object of This Class or Interface
logical constraints	IloOr, IloAnd, and methods such as <code>not</code>

For an explanation of quadratic constraints, see *Solving problems with quadratic constraints (QCP)*. For more information about quadratic objective functions, see *Solving problems with a quadratic objective (QP)*. For examples of piecewise linear constraints, see *Using piecewise linear functions in optimization: a transport example*. For a description of special ordered sets, see *Using special ordered sets (SOS)*. For more about logical constraints, see *Logical constraints in optimization*.

Using IloModeler

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

Variables in a model

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

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

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

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

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

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

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

Expressions

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

```
x1 + 2*x2
```

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

```
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 can be edited, especially convenient when linear expressions are being built in a loop, like this:

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

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 that 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*.

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 =, ≤, or ≥. The following table shows how to choose lb and ub for modeling these relations:

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

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

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

Again, all constructors for ranged constraints come in pairs, one constructor with and one without an argument for the name.

The objective function

The objective function in Concert Technology is represented by an object 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:

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 are created as explained in *Using IloModeler*; then these components must be added to the active model. The active model is the model implemented by the `IloCplex` object itself. In fact, `IloModeler` is an extension of the `IloModel` interface defining the model API. Thus, `IloCplex` implements `IloModel`, or in other words, an `IloCplex` object is a model. The model implemented by the `IloCplex` object itself is referred to as the active model of the `IloCplex` object, or if there is no possibility of confusion between several optimizers, simply as the active model.

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

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

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

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

is equivalent to:

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

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

Building the model

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

The example includes a set of foods, where food j has a unit cost of `foodCost[j]`. The minimum and maximum amount of food j which can be used in the diet is designated `foodMin[j]` and `foodMax[j]`, respectively. Each food j also has a nutritional value `nutrPerFood[i][j]` for all possible nutrients i . The nutritional requirement states that in the diet the amount of every nutrient i consumed must be within the bounds `nutrMin[i]` and `nutrMax[i]`.

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

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

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

```
nutrMin[i]  $\leq$   $\sum_j$  nutrPerFood[i][j] * Buy[j]  $\leq$  nutrMax[i]
```

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

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

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

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

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

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

```
        model.scalProd(data.nutrPerFood[i], Buy),
        data.nutrMax[i]);
    }
}
```

The function accepts several arguments. The argument `model` is used for two purposes:

- ◆ 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, the argument `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 the 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 the 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 the interface `IloModeler` rather than `IloCplex`. This convention allows the function to be called without change in another implementation of `IloModeler`, such as `IloCP`.

Solving the model

After you have created an optimization problem in your active model, you solve it by means of the `IloCplex` object. For an object named `cplex`, for example, you solve by calling the method like this:

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

The `solve` method returns a Boolean value specifying 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 reached 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 *Solution status*.

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 has been found with the `solve` method, you access it and then query it using a variety of methods. The objective function can be accessed by the call:

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

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

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

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

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

Similarly, you query slack values for the constraints in the active model by means of the methods `IloCplex.getSlack` or `IloCplex.getSlacks`.

These ideas apply to solving and printing the solution to the diet problem as well.

```
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` optimizes the active model and, upon success, the application prints solution information.

Choosing an optimizer

Describes algorithms implemented in the Java API for solving optimization problems.

In this section

Overview

Introduces the Java methods that solve a problem in ILOG CPLEX.

What does CPLEX solve?

Explains what is solved in terms of problem types.

Solving a single continuous model

Shows how to choose the optimizer to solve a single continuous model.

Solving subsequent continuous relaxations in a MIP

Shows how to choose the optimizer to solve a series of relaxations.

Overview

The algorithm implemented in the `solve` methods can be controlled and if necessary tailored to the particular needs of the model. The most important control you exercise is your selection of the optimizer.

What does CPLEX solve?

Given an 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*, *Solving LPs: barrier optimizer*, and *Solving network-flow problems* in this manual.

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

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

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

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

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

`IloCplex` solves QCP models using the barrier optimizer. For more about this kind of problem, see *Solving problems with quadratic constraints (QCP)*, 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)*.

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. *Algorithm types for RootAlg* shows you the available types of algorithms.

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`.

In other words, use the root algorithm parameter to choose the optimizer for solving at the root, and use the node algorithm parameter to choose the optimizer for solving at the subsequent nodes of the problem.

The algorithm choices appear in *Algorithm types for NodeAlg*

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

Describes the features that control optimizers in the Java API.

In this section

Overview

Introduces the methods to invoke ILOG CPLEX parameters in a Java application.

Parameters

Introduces parameters as a means to control optimizers in the Java API.

Priority orders and branching directions

Introduces methods in the Java API to specify priority orders and branching directions.

Overview

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 *Constants in IloCplex.PrimalPricing* for better maintainability of the code.

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);
```

Classes with parameters defined by integers. gives an overview of the classes defining constants for parameters.

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>

Class	For use with parameters:
<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 these:

- ◆ `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*. Control callbacks are discussed in *Using optimization callbacks*. (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* contrasts the advantages of both.

More solution information

Describes additional information available from a solution.

In this section

Overview

Introduces methods in Java to access information about a solution in ILOG CPLEX.

Writing solution files

Describes methods of the Java API to write solution files.

Dual solution information

Describes methods of the Java API to access dual solution information.

Basis information

Describes methods of the Java API to access basis information.

Infeasible solution information

Describes methods of the Java API to access information about an infeasible solution.

Solution quality

Describes methods of the Java API to access information about the quality of a solution.

Overview

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.

Writing solution files

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

After solving, you can call the method `IloCplex.writeMIPstart` to write a MIP solution suitable for a restart. The file it writes is in MST format. That format is documented by *MST file format: MIP starts* 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 you decide 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 investigating 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*.

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* 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 `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

Introduces the Java class supporting matrix-oriented modeling in ILOG CPLEX.

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 ≤ Ax ≤ U
with these bounds L ≤ x ≤ 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: $L \quad Ax \quad U$

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

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

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

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

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

Modeling by column

Introduces modeling by columns as implemented in the Java API.

In this section

What is modeling by column?

Explains modeling by column.

Procedure for Modeling by Column

Tells how to model by columns in the Java API.

What is 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 *Procedure for Modeling by Column*.

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`.

After you have created the proper column object, use it 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 this 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++*
- ◆ *Example: optimizing the diet problem in C#.NET*
- ◆ *Example: optimizing the diet problem in the Callable Library*

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 deciding 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.

After 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);
```

The 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 an argument 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 like this:

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

After the model has been 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 `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 `plex.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

Explores the features that ILOG CPLEX offers to users of C#.NET through Concert Technology.

In this section

Prerequisites

Prepares the reader for this tutorial.

Describe

States the aim of the tutorial and describes the finished application.

Step 1: Describe the problem

Suggests questions to ask when you describe the problem.

Step 2: Open the file

Tells where to find lessons for this tutorial.

Model

Describes the modeling step of this tutorial.

Step 3: Create the model

Shows lines to create the model in a .NET application of ILOG CPLEX.

Step 4: Create an array to store the variables

Shows lines to create an array in a .NET application of ILOG CPLEX.

Step 5: Specify by row or by column

Shows lines to add to accept arguments in a .NET application of ILOG CPLEX.

Build by Rows

Introduces reason for a static method in this example.

Step 6: Set up rows

Shows lines to declare a static method in a .NET application of ILOG CPLEX.

Step 7: Create the variables: build and populate by rows

Shows lines to create variables for this example.

Step 8: Add objective

Shows lines to add an objective to this example.

Step 9: Add nutritional constraints

Shows lines to add constraints to this example.

Build by Columns

Introduces the need for a static method in the example.

Step 10: Set up columns

Shows lines to build the model by columns.

Step 11: Add empty objective function and constraints

Shows lines to add objective function and constraints in a model built by columns.

Step 12: Create variables

Shows lines to create variables in a model built by columns.

Solve

Describes steps in the tutorial to solve the problem.

Step 13: Solve

Shows line to invoke the solver in this example.

Step 14: Display the solution

Shows lines to display the solution in this example.

Step 15: End application and free license

Shows a line to end the application and free the license for this example.

Good programming practices

Describes good programming practices to include in the application.

Step 16: Read the command line (data from user)

Shows lines to read data entered by a user of this example.

Step 17: Show correct use of command line

Shows lines that prompt user about correct usage.

Step 18: Enclose the application in try catch statements

Shows lines to enclose this application in try catch statements.

Example: optimizing the diet problem in C#.NET

Tells where to find complete application, project, lessons for the tutorial.

Prerequisites

This tutorial walks you through an application based on the widely published diet problem.

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 to 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 deciding 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: Specify by row or by column

Go to the comment Step 5 in `DietLesson.cs`, and add the following lines to specify 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*, 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 application 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\system\format\Diet.csproj`

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

`yourCPLEXhome\examples\tutorials\Dietlesson.cs`

ILOG CPLEX Callable Library

Shows how to write C applications using the ILOG CPLEX Callable Library.

In this section

Architecture of the ILOG CPLEX Callable Library

Describes the architecture of the Callable Library C API.

Using the Callable Library in an application

Describes an application of the C API.

ILOG CPLEX programming practices

Lists some of the programming practices ILOG observes in developing and maintaining the ILOG CPLEX Callable Library (C API).

Managing parameters from the Callable Library

Introduces routines to manage parameters of ILOG CPLEX from the C API.

Example: optimizing the diet problem in the Callable Library

Walks through an example applying the C API.

Using surplus arguments for array allocations

Describes coding conventions of the C API to manage surplus arguments and array allocations.

Example: using query routines lpex7.c

Shows how to use query routines of the C API in an example.

Architecture of the ILOG CPLEX Callable Library

Describes the architecture of the Callable Library C API.

In this section

Overview

Offers general remarks and a diagram of the architecture of a Callable Library application.

Licenses

Describes licensing conventions for the C API.

Compiling and linking

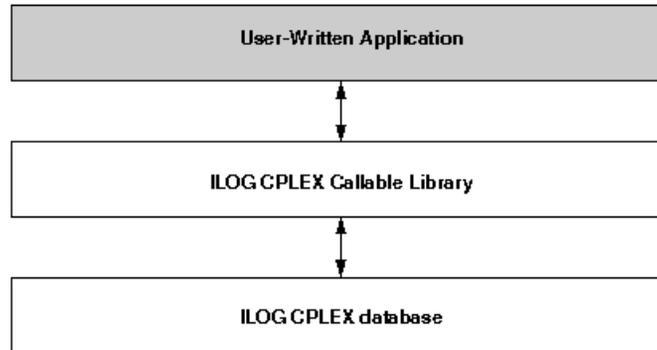
Tells where to find information about compiling and linking with the C API.

Overview

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 *A view of the ILOG CPLEX Callable Library*. 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.



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

Describes an application of the C API.

In this section

Overview

Outlines steps to include in a Callable Library application.

Initialize the ILOG CPLEX environment

Tells how to initialize the environment in the C API.

Instantiate the problem as an object

Tells how to instantiate a problem object in the C API.

Put data in the problem object

Tells how to populate a problem object with data in the C API.

Optimize the problem

Tells how to optimize a problem in the C API.

Change the problem object

Tells how to modify a problem in the C API.

Destroy the problem object

Tells when to destroy a problem object in the C API.

Release the ILOG CPLEX environment

Tells how to end an application of the C API.

Overview

This topic 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

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.

Note: An attempt to use a problem object in any environment other than the environment (or a child of that environment) where the problem object was created will raise an error.

Instantiate the problem as an object

After 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* 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*.

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* 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 `CPXchgbd`s. 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

Lists some of the programming practices ILOG observes in developing and maintaining the ILOG CPLEX Callable Library (C API).

In this section

Overview

Introduces general characteristics of variables and routines in the Callable Library.

Variable names and calling conventions

Describes the coding practices of the C API with respect to variable names and calling conventions.

Data types

Describes the data types available in the C API.

Ownership of problem data

Describes ownership conventions in the C API with respect to data.

Problem size and memory allocation issues

Describes conventions of the C API with respect to size of problem and memory allocation.

Status and return values

Describes coding conventions of the C API with respect to status and return values.

Symbolic constants

Describes coding conventions about symbolic constants in the C API.

Parameter routines

Describes routines of the C API to control ILOG CPLEX parameters.

Null arguments

Describes coding conventions of the C API with respect to null arguments.

Row and column references

Describes coding conventions of the C API with respect to indexing arrays.

Character strings

Describes coding conventions of the C API with respect to character strings.

Problem data

Describes routines to verify problem data and diagnose bugs in an application of the C API.

Callbacks

Describes coding conventions of the C API with respect to callbacks.

Portability

Describes coding conventions of the C API to make portable applications.

FORTRAN interface

Describes coding conventions of the C API to support interface with a FORTRAN application.

C++ interface

Describes conventions of the C API for interface with C++ applications.

Overview

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 re-entrant 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* 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 the table *Special data types in the ILOG CPLEX Callable Library*. The types starting with CPXC represent the corresponding pointers to constant (const) objects.

Special data types in the ILOG CPLEX Callable Library

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

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

Ownership of problem data

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

Problem size and memory allocation issues

As indicated in *Change the problem object*, 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 set 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) `plex.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. *Callable Library routines for parameters in the ILOG CPLEX environment* shows you the names and purpose of these routines. Each of these routines accepts the same first argument: a pointer to the ILOG CPLEX environment (that is, the pointer returned by `CPXopenCPLEX`). The second argument of each of those parameter routines is the parameter number, a symbolic constant defined in the header file, `cplex.h`. *Managing parameters from the Callable Library* offers more details about parameter settings.

Callable Library routines for parameters in the ILOG CPLEX environment

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

Null arguments

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

Row and column references

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

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

- ◆ 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 reveal 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.

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 *data consistency checking switch* parameter `CPX_PARAM_DATACHECK` (int) to the value `CPX_ON` to activate additional checking of array arguments for `CPXcopy Data`, `CPXread Data`, and `CPXchg Data` routines (where *Data* varies). The additional checks include:

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

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 *messages to screen switch* 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 node processing, 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 optimization callbacks explores callback facilities in greater detail, and *Advanced MIP control interface* discusses control callbacks in particular.

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* and *Example: Callable Library message channels*.

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* 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 learn 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 the *clock type for computation time* parameter `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

Walks through an example applying the C API.

In this section

Overview

Describes the problem in the example.

Problem representation

Describes how to represent the problem by means of the C API.

Program description

Describes the architecture of an application from the C API solving the diet problem.

Solving the model with CPXlpopt

Shows the routine used to solve the problem.

Complete program

Tells where to find the C implementation of the diet problem online.

Overview

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. The example `diet.c` illustrates these points.

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 decide 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:

```
nutrmin[i] ... nutrmax[i]
```

where ... is left to be filled after 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:

```
IloObjective obj = cplex.objective(sense, expr, name);
cplex.setParam(IloCplex.IntParam.PPriInd,
               IloCplex.PrimalPricing.Steep);
IloColumn col = cplex.column(obj, 1.0).and(cplex.column(rng, 2.0));
cost(foodCost[j]) "+" "sum_i" (rng[i] (nutrper[i][j]))
```

where the notation "+" and "sum" indicates that you “add” the new variable j to the objective `cost` and constraints `rng[i]`. The value in parentheses is the linear coefficient that is used for the new variable.

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 the file `<ilcplex/cplex.h>` at the beginning of the application. After a number of lines that establish the calling sequences for the routines that are to be used, the `main` function of the application 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 set at run time by an argument on the command line. The routine `populatebyrow` operates by first defining all the columns through a call to `CPXnewcols` and then repeatedly calls `CPXaddrows` to enter the data of the constraints. The routine `populatebycolumn` takes the complementary approach of establishing all the rows first with a call to `CPXnewrows` and then sequentially adds the column data by calls to `CPXaddcols`.

Solving the model with CPXlpopt

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

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

Complete program

The complete program, `diet.c`, appears 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 discover 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 discover 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 set the length of the array `cmatbeg`. The array `cmatbeg` has one element for each column in the requested range. Since this length is known ahead of time, your application does not need to call a query routine to calculate it. More generally, query routines use surplus arguments in this way only for the length of any array required to store problem data of unknown length. Problem data in this category include nonzero matrix entries, row and column names, other problem data names, special ordered sets (SOS), priority orders, and MIP start information.

Example: using query routines lpex7.c

This example uses the ILOG CPLEX Callable Library query routine `CPXgetcolname` to get the column names from a problem object. To do so, it applies the programming pattern just outlined in *Using surplus arguments for array allocations*. 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 discovers 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*.

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:

In this section

Developing CPLEX applications

Offers suggestions for improving application development and debugging completed applications.

Managing input and output

Describes input to and output from ILOG CPLEX.

Timing interface

Introduces the timing interface available in ILOG CPLEX

Licensing an application

Describes ILOG CPLEX runtime and development licensing conventions.

Tuning tool

The *tuning tool*, a utility to aid you in improving the performance of your optimization applications, analyzes a model or a group of models and suggests a suite of parameter settings for you to use that provide better performance than the default parameter settings for your model or group of models. This chapter documents the tuning tool.

Developing CPLEX applications

Offers suggestions for improving application development and debugging completed applications.

In this section

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.

Using the Interactive Optimizer for debugging

Describes Interactive Optimizer more fully as debugger.

Eliminating common programming errors

Serves as a checklist to eliminate common pitfalls from an application.

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.

In this section

Prototype the model

Describes use of prototypes.

Identify routines to use

Describes purpose of decomposition.

Test interactively

Introduces Interactive Optimizer as debugger.

Assemble data efficiently

Describes populating the problem with data.

Test data

Presents a data checking tool.

Test and debug the model

Introduces the preprocessor, conflict refiner, and FeasOpt as testing and debugging tools..

Choose an optimizer

Describes optimizer choice in terms of problem type.

Program with a view toward maintenance and modifications

Suggests practices to facilitate maintenance and modification of applications.

Prototype the model

Begin by creating a small-scale version of the model for your problem. 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 deciding 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 topics, 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`, `CPXaddrows`, `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 a parameter to check the correctness of data used in problem creation and problem modification methods: the *data consistency checking switch* (`DataCheck` or `CPX_PARAM_DATACHECK`). When this parameter is set, ILOG CPLEX will perform extra checks to confirm 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.

Test and debug the model

The optimizers available in ILOG CPLEX work on primarily on models for which bounded feasible solutions exist. For unbounded or infeasible models, that is, for models for which no bounded, feasible solution exists, ILOG CPLEX offers tools to enable you to test and debug the model.

Early reports of infeasibility based on preprocessing reductions explains how to interpret reports from the preprocessor that your model is infeasible or unbounded. It also describes how to use a parameter to further your analysis of those preprocessor reports.

Diagnosing infeasibility by refining conflicts explains the conflict refiner, a tool that can help you diagnose incompatibilities among constraints within your model or MIP start.

Repairing infeasibilities with FeasOpt introduces a tool that attempts to repair infeasibility in a model by modifying the model according to preferences that you express.

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* and *Solving problems with a quadratic objective (QP)* offer more detail about improving performance with respect to these problems. Integer programming problems are more sensitive to specific parameter settings, so you may need to experiment with them, as suggested in *Solving mixed integer programming problems (MIP)*.

In either case, the Interactive Optimizer in ILOG CPLEX lets you try different parameter settings and different optimizers to decide 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* 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 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, after it works correctly, look for opportunities to improve performance.

Debug effectively

Using diagnostic routines for debugging, 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* 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 `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* 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 discover 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

Serves as a checklist to eliminate common pitfalls from an application.

In this section

Check your include files

Identifies essential include file (header file).

Clean house and try again

Suggests recovery strategy.

Read your messages

Introduces warning and error messages.

Check return values

Explains purpose of return values.

Beware of numbering conventions

Describes indexing and numbering conventions.

Make local variables temporarily global

Describes investigation of the heap through global variables.

Solve the problem you intended

Recommends interactive debugging and diagnostic routines.

Special considerations for FORTRAN

Describes FORTRAN conventions for indexing, numbering.

Tell us

Tells where to report problems.

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 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 *messages to screen switch* `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*, and the diagnostic routines, as in *Using diagnostic routines for debugging*.

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

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, C++, Java, and other languages 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

Describes input to and output from ILOG CPLEX.

In this section

Overview

Provides background for managing input and output for ILOG CPLEX.

Understanding file formats

Explains programming considerations about widely used file formats.

Using Concert XML extensions

Describes facilities for serialization of models and solutions.

Using Concert csvReader

Describes facilities for reading comma separated values (CSV).

Managing log files

Describes facilities for working with log files.

Controlling message channels

Describes message channels.

Overview

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

Explains programming considerations about widely used file formats.

In this section

Overview

Introduces the reference manual about file formats supported by ILOG CPLEX.

Working with LP files

Describes programming considerations for working with LP file format.

Working with MPS files

Describes programming considerations for working with MPS file format.

Converting file formats

Describes programming considerations about converting file formats.

Overview

The reference manual *ILOG CPLEX File Formats* documents the file formats that ILOG CPLEX supports more fully. The following topics cover programming considerations about widely used file formats.

Working with LP files

LP files are row-oriented so you can look at a problem as you enter it in a naturally and intuitively algebraic way. However, ILOG CPLEX represents a problem internally in a column-ordered format. This difference between the way ILOG CPLEX accepts a problem in LP format and the way it stores the problem internally may have an impact on memory use and on the order in which variables are displayed on screen or in files.

Variable order and LP files

As ILOG CPLEX reads an LP format file by rows, it adds columns as it encounters them in a row. This convention will have an impact on the order in which variables are named and displayed. For example, consider this problem:

```
Maximize          2x2 + 3x3

subject to

      -x1 +   x2 + x3   20

      x1  - 3x2 + x3   30

with these bounds

      0          x1    40

      0          x2    +

      0          x3    +
```

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 *Options for the convert utility and corresponding file extensions*, 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.

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 *Concert Technology C++ Reference Manual*.

◆ `IloCsvReader`

An object of this class is capable of reading data from a CSV file and passing the data to your application. There are methods in this class for recognizing the first line of the file as a header, for indicating whether or not to cache the data, for counting columns, for counting lines, for accessing lines by number or by name, for designating special characters, for indicating separators, and so forth.

◆ `IloCsvLine`

An object of this class represents a line of a CSV file. The constructors and methods of this class enable you to designate special characters, such as a decimal point, separator, line ending, and so forth.

◆ `IloCsvReader::Iterator`

An object of this embedded class is an iterator capable of accessing data in a CSV file line by line. This iterator is useful, for example, in programming loops of your application, such as `while` -statements.

Managing log files

Describes facilities for working with log files.

In this section

Overview

Introduces log files from ILOG CPLEX.

Creating, renaming, relocating log files

Describes methods for creating, renaming, and relocating log files.

Closing log files

Describes facilities for closing log files.

Overview

As ILOG CPLEX is working, it can record messages to a log file. By default, the Interactive Optimizer creates the log file in the directory where it is running, and it names the file `cplex.log`. If such a file already exists, ILOG CPLEX adds a line indicating the current time and date and then appends new information to the end of the existing file. That is, it does not overwrite the file, and it distinguishes different sessions within the log file. By default, there is no log file for Component Library applications.

You can locate the log file where you like, and you can rename it. Some users, for example, like to create a specifically named log file for each session. Also you can close the log file in case you do not want ILOG CPLEX to record messages to its default log file.

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

Describes message channels.

In this section

Overview

Introduces message channels for ILOG CPLEX.

Parameter for output channels

Describes control for message channels in Interactive Optimizer and C API.

Callable Library routines for message channels

Describes routines to control message channels in the C API.

Example: Callable Library message channels

Demonstrates control of message channels in the C API.

Concert Technology message channels

Describes control of message channels in Concert Technology C++ API.

Overview

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 *Options for the output channel command*. 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;*
- ◆ *Callable Library routines for message channels;*
- ◆ *Example: Callable Library message channels;*
- ◆ *Concert Technology message channels.*

Parameter for output channels

Besides the log-file parameter, the 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`. *Options for the output channel command* summarizes the information carried over each channel.

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. *Channels directing output to the screen or to a file* 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`.

Channels directing output to the screen or to a file

Channel	Default value 1	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

Channel	Default value 1	Meaning
results	y	blank directs output to screen and to a file
warnings	y	blank directs output to screen and to a file

Callable Library routines for message channels

Interactive Optimizer and the Callable Library define several message channels for flexible control over message output:

- ◆ `cpxresults` for messages containing status and progress information;
- ◆ `cpxerror` for messages issued when a task cannot be completed;
- ◆ `cpxwarning` for messages issued when a nonfatal difficulty is encountered; or when an action taken may have side-effects; or when an assumption made may have side-effects;
- ◆ `cpxlog` for messages containing information that would not conventionally be displayed on screen but could be useful in a log file.

Output messages flow through message channels to destinations. Message channels are associated with destinations through their destination list. Messages from routines of the ILOG CPLEX Callable Library are assigned internally to one of those predefined channels. Those default channels are C pointers to ILOG CPLEX objects; they are initialized by `CPXopenCPLEX`; they are not global variables. Your application accesses these objects by calling the routine `CPXgetchannels`. You can use these predefined message channels for your own application messages. You can also define new channels.

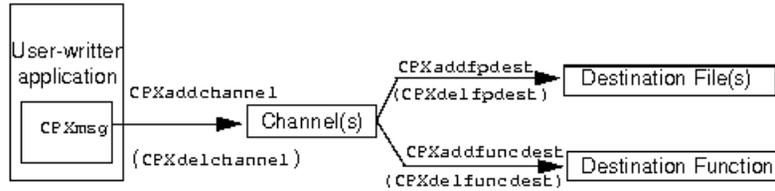
An application using routines from the ILOG CPLEX Callable Library produces no output messages unless the application specifies message handling instructions through one or more calls to the message handling routines of the Callable Library. In other words, the destination list of each channel is initially empty.

Messages from multiple channels may be sent to one destination. All predefined ILOG CPLEX channels can be directed to a single file by a call to `CPXsetlogfile`. Similarly, all predefined ILOG CPLEX channels except `cpxlog` can be directed to the screen by the *messages to screen switch* `CPX_PARAM_SCRIND`. For a finer level of control, or to define destinations for application-specific messages, use the following message handling routines, all documented in the ILOG CPLEX Reference Manual:

- ◆ `CPXmsg` 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;

After channel destinations are established, messages can be sent to multiple destinations by a single call to a message-handling routine.



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.

The complete program `lpex5.c` appears online in the standard distribution at *yourCPLEXinstallation/examples/src*. This example derives from `lpex1.c`, a program in *ILOG CPLEX Getting Started*. 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.

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.

Timing interface

Introduces the timing interface available in ILOG CPLEX

In this section

Using the timing interface

Describes the timing interface for accessing time stamps.

Using the timing interface

There are methods and routines in ILOG CPLEX that provide a time stamp to enable you to measure computational time. For example, an application can invoke one of these methods or routines at the beginning and end of an operation; then compare the two time stamps to compute elapsed time in seconds.

- ◆ In Concert Technology
 - In the C++ API, `IloCplex::getCplexTime` returns a time stamp that applications may use to calculate elapsed time in seconds.
 - In the Java API, `IloCplex.getCplexTime` returns a time stamp that applications may use to calculate elapsed time in seconds.
 - In the .NET API, the property `Cplex.CplexTime` accesses a time stamp that applications may use to calculate elapsed time in seconds.
- ◆ In the Callable Library, `CPX_gettime` returns a time stamp that applications may use to calculate elapsed time in seconds.

In addition, other methods and routines return a time stamp specifying when a time limit will occur; this time stamp may be used in calculations with the time stamp returned by the previous methods or routines to compute remaining time in seconds from **callbacks**.

- ◆ In Concert Technology
 - In the C++ API, `CallbackI::getEndTime` returns a time stamp specifying when a time limit will occur; this time stamp may be used in calculations with the time stamp returned by `getTime` to compute remaining time in seconds.
 - In the Java API, `CpxCallback.getEndTime` returns a time stamp specifying when a time limit will occur; this time stamp may be used in calculations with the time stamp returned by `getTime` to compute remaining time in seconds.
 - In the .NET API, the property `Cplex.Callback.EndTime` accesses a time stamp specifying when a time limit will occur; this time stamp may be used in calculations with the time stamp accessed by `CplexTime` to compute remaining time in seconds.
- ◆ In the Callable Library, `CPX_CALLBACK_INFO_ENDTIME` is a symbolic value that `CPX_getcallbackinfo` can supply in its argument `whichinfo`. That value is a time stamp of the point in time when the optimization will terminate if optimization does not finish before that point. In other words, this symbol is useful in measuring time through information callbacks.

For a sample of these timing features, see these examples among those distributed with the product in `yourCPLEXinstallation/examples`:

- ◆ `ilomipex4.cpp` in C++ in Concert Technology
- ◆ `MIPex4.java` in Java in Concert Technology
- ◆ `MIPex4.cs` in C#.NET in Concert Technology
- ◆ `MIPex4.vb` in Visual Basic.NET in Concert Technology
- ◆ `mipex4.c` in C in the Callable Library

Licensing an application

Describes ILOG CPLEX runtime and development licensing conventions.

In this section

Overview

Outlines general considerations about licensing an application that uses ILOG CPLEX.

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.

Routines and methods for licensing

Describes routines of the C API and methods of Concert Technology for licensing an application of ILOG CPLEX.

Examples

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

Summary

Summarizes licensing conventions for applications of ILOG CPLEX.

Overview

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.

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.

In this section

File-based RTNODE, RTSTOKEN or TOKEN keys

Describes file-based ILM licenses.

Memory-based RUNTIME keys

Describes memory-based ILM licenses.

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 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.

In this section

CPXputenv routine for C and C++ users

Illustrates opening the CPLEX environment for C and C++ users.

The putenv method for Java users

Illustrates putenv method for Java users.

The Putenv method for .NET users

Illustrates Putenv method for .NET users.

CPXRegisterLicense routine for C and C++ users

Illustrates license registration for C and C++ users.

The registerLicense method for Java users

Illustrates license registration for Java users.

The RegisterLicense method for .NET users

Illustrates license registration for .NET users.

CPXputenv routine for C and C++ users

```
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);
```

Note: 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

```
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

```
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

```
static char *ilm_license=\
"LICENSE ILOG Incline\n\
  RUNTIME CPLEX 11.200 21-Jul-2011 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

```
static String ilm_CPLEX_license=
"LICENSE ILOG Test\n RUNTIME CPLEX 11.200 021-Jul-2015 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

```
internal static string ilm_CPLEX_license="LICENSE ILOG User\n RUNTIME CPLEX  
  
11.200 05-Aug-2015 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.

Tuning tool

The *tuning tool*, a utility to aid you in improving the performance of your optimization applications, analyzes a model or a group of models and suggests a suite of parameter settings for you to use that provide better performance than the default parameter settings for your model or group of models. This chapter documents the tuning tool.

In this section

Meet the tuning tool

Introduces the tuning tool.

Invoking the tuning tool

Describes methods and routines to invoke the tuning tool.

Example: time limits on tuning in the Interactive Optimizer

Shows the effect of time limits on the tuning tool in the Interactive Optimizer.

Fixing parameters and tuning multiple models in the Interactive Optimizer

Shows options of the tuning tool command in the Interactive Optimizer.

Tuning models in the Callable Library (C API)

Describes the tuning tool in the C API.

Callbacks for tuning

Describes callbacks available to the tuning tool.

Terminating a tuning session

Describes facilities to end a tuning session.

Meet the tuning tool

Introduces the tuning tool.

In this section

Overview

Defines the scope of the tuning tool.

If ILOG CPLEX solves your problem to optimality

Distinguishes problems solved to optimality.

If ILOG CPLEX finds solutions but does not prove optimality

Distinguishes problems with solutions not proven optimal.

Tuning and time limits

Introduces parameters to set time limits on tuning.

Tuning results

Describes typical results of a tuning session.

Overview

The tuning tool looks at parameters to improve *solving time*. If your model suffers from numeric instability, the tuning tool will not attempt to correct that problem. Furthermore, if there is insufficient memory to accommodate optimization of your model, the tuning tool will not correct that problem either. In short, the tuning tool does not magically eliminate all performance bottlenecks. However, if you understand the performance issues of your model, the tuning tool can help you discern parameter settings that lead to faster solving time.

The recommended practice with the tuning tool is to solve your model first with default settings and to consider the results before invoking the tuning tool. Your analysis of those results directs you toward the next step. The following sections sketch typical scenarios for working effectively with the tuning tool and outline what to expect from the tuning tool.

If ILOG CPLEX solves your problem to optimality

If ILOG CPLEX solves your problem to optimality, you may still want to apply the tuning tool to discover whether you can solve the model faster. In such a case, bear in mind that the tuning tool performs several optimization runs as it goes about its work. These optimization runs may take six to eight times longer than the default run that produced your optimal results. If that projected time (six to eight times longer than the initial default run) seems too long for your purpose, then consider setting a general time limit by means of the *optimizer time limit* parameter (`TiLim`, `CPX_PARAM_TILIM`) or consider setting a specific tuning time limit per problem, per optimization, by means of the *tuning time limit* parameter (`TuningTiLim`, `CPX_PARAM_TUNINGTILIM`). *Example: time limits on tuning in the Interactive Optimizer* illustrates this approach through time limits more fully.

If ILOG CPLEX finds solutions but does not prove optimality

In the case where ILOG CPLEX finds solutions for your model but does not prove optimality in your initial run before invoking the tuning tool, you will likely want to set the time limit per model. See *Tuning and time limits* for more about that idea.

In situations where ILOG CPLEX does not solve your model to optimality for a reason other than a time limit, you should address that reason before you apply the tuning tool.

For example, if your initial run results in an out-of-memory error, consider setting the memory emphasis parameter (*memory reduction switch*: `MemoryEmphasis`, `CPX_PARAM_MEMORYEMPHASIS`). Then create a file in which you specify a fixed setting of that parameter for the tuning tool to respect. Pass that file to the tuning tool with your model. *Fixing parameters and tuning multiple models in the Interactive Optimizer* illustrates this approach.

Tuning and time limits

The tuning process is affected by two time limit parameters.

The overall time limit for the tuning process is set with the general time limit parameter (*optimizer time limit*: `TiLim`, `CPX_PARAM_TILIM`).

Tuning consists of a series of optimizations of each of the problems to be tuned. The time limit on each of these optimizations is specified by the tuning time limit parameter (*tuning time limit*: `TuningTiLim`, `CPX_PARAM_TUNINGTILIM`). The default value of this parameter is set to a value much smaller than the default value of the overall time limit, so as to avoid runs of indeterminate length. This parameter governs complete optimizations started within the tuning tool. The tuning tool may also launch shorter and partial optimizations; in such cases, it uses this parameter as guidance to compute shorter time limits.

The per-problem, per-optimization time limit can also be set using the general time limit parameter (*optimizer time limit*: `TiLim`, `CPX_PARAM_TILIM`) in a fixed parameter set. *Fixing parameters and tuning multiple models in the Interactive Optimizer* explains more about that approach. You would want to set the per-problem, per-optimization time limit in this way if you are using a time limit in your usual (non-tuning) runs of the models.

Tuning results

The tuning tool selects the best suite of parameter settings based on the performance times of the several optimizations of the model or models. Because the performance times cannot be measured exactly, the results of tuning can differ from one tuning run to another if the times for the individual optimizations are close.

Invoking the tuning tool

The tuning tool is available in all components of ILOG CPLEX: Concert Technology, Callable Library, and Interactive Optimizer.

In Concert Technology, you invoke the tuning tool through these methods:

- ◆ `tuneParam` in the C++ API; see also the example `ilotuneset.cpp`;
- ◆ `tuneParam(String, String)` in the Java API; see also the example `TuneSet.java`;
- ◆ `Cplex.TuneParam` in the .NET API; see also the example `TuneSet.cs` or `TuneSet.vb`.

The tool is also part of the Callable Library (C API) through separate routines for tuning a single model or a group of models. For more detail about the C routine for tuning a single model or a group of models, see *Tuning models in the Callable Library (C API)*. For a sample application, see `examples/src/cplex/tuneset.c`.

In the Interactive Optimizer, you can tune one model or a group of models with the `tune` command. See the *Example: time limits on tuning in the Interactive Optimizer*.

You can specify whether you want the least worst performance or the best average performance across a set of models with the tuning measure parameter, (*tuning measure*: `TuningMeasure`, `CPX_PARAM_TUNINGMEASURE`), documented in the reference manual *ILOG CPLEX Parameters*.

When you are tuning a single model, you can ask ILOG CPLEX to permute the model and re-tune to get more robust results, as explained in the documentation of the repeat tuning parameter (*tuning repeater*: `TuningRepeat`, `CPX_PARAM_TUNINGREPEAT`).

You can also set a time limit specific to tuning per problem and per optimization run, as documented for the tuning time limit parameter (*tuning time limit*: `TuningTiLim`, `CPX_PARAM_TUNINGTILIM`). *Example: time limits on tuning in the Interactive Optimizer* shows how to set such a time limit.

Example: time limits on tuning in the Interactive Optimizer

As an example, suppose that you want to spend an overall amount of time tuning the parameter settings for a given model, say, 1000 seconds. Also suppose that you want ILOG CPLEX to make multiple attempts within that overall time limit to tune the parameter settings for your model. Suppose further that you want to set a time limit on each of those attempts, say, 200 seconds per attempt.

In the Interactive Optimizer, first enter your model into a session (for example, by reading a formatted file), like this:

```
read model.mps
```

Then set the overall time limit through the environment with this command:

```
set timelimit 1000
```

That command makes 1000 seconds available as a resource in the environment. The command effectively sets the overall time limit (*optimizer time limit*: `TiLim`, `CPX_PARAM_TILIM`).

Then set the tuning time limit, like this:

```
set tune timelimit 200
```

This series of commands tells ILOG CPLEX to tune the parameters of your model, making multiple attempts of 200 seconds each (set by the *tuning time limit*: `TuningTiLim`, `CPX_PARAM_TUNINGTILIM`), within an overall time limit of 1000 seconds (set by the *optimizer time limit*: `TiLim`, `CPX_PARAM_TILIM`).

Fixing parameters and tuning multiple models in the Interactive Optimizer

Shows options of the tuning tool command in the Interactive Optimizer.

In this section

Invoking the tuning tool in the Interactive Optimizer

Shows commands to invoke the tuning tool in the Interactive Optimizer.

Fixed parameters to respect

Describes facilities in Interactive Optimizer to exclude parameters from consideration by the tuning tool.

Files of models to tune

Describes facility in the Interactive Optimizer to tune multiple models simultaneously.

Invoking the tuning tool in the Interactive Optimizer

The command to invoke the tuning tool in the Interactive Optimizer is: `tune`

Optionally, that command may take one or two arguments, in either order, like this:

```
tune paramfile.prm modelfile
```

or

```
tune modelfile paramfile.prm
```

The following sections explain those optional arguments.

Fixed parameters to respect

If you supply an optional file name with the extension `.prm`, that file contains the formatted list of parameters and their values that you do not want ILOG CPLEX to tune. That is, you want ILOG CPLEX to respect those parameter settings. If you do not supply a parameter file to the `tune` command, then all parameters are subject to tuning.

For example, if you have a model that you know has numerical issues, and you have already determined that you want to set the numerical emphasis parameter yourself, then a PRM file suitable for tuning your model would contain the following heading and parameter setting:

```
CPLEX Parameter File Version 11.2.0
CPX_PARAM_NUMERICAL EMPHASIS      1
```

Tip: A PRM file must have a correctly formatted header to let CPLEX know which version to expect.

An easy way to generate a PRM file with a correct heading and fixed parameter settings is to read your model, set the parameters you wish, and write a PRM file from the Interactive Optimizer, with this command:

```
write filename .prm
```

Files of models to tune

If you supply an optional file name, such as `modelfile`, that file contains a list of files, one file name per line. Each file contains a model to tune. Each file must specify its type, such as `.mps`, `.lp`, or `.sav`, as the extension in the file name. Optionally, those cited files may be compressed, as specified by the extension `.gz` or `.bz2`. Here is an example of the contents of such a file, specifying three compressed files of type `.mps` available in the current working directory:

```
p0033.mps.gz  
p0548.mps.gz  
p2756.mps.gz
```

For models not in the current working directory, you can specify a path to them with the usual syntax for your platform.

ILOG CPLEX will uncompress the model files, read them according to their type of format, and apply the tuning tool to that suite of files. If you do not specify a list of models to tune, then ILOG CPLEX tunes the model currently in the Interactive Optimizer.

Tuning models in the Callable Library (C API)

The routine `CPXtuneparam` tunes one existing model, and the routine `CPXtuneparamprobset` tunes a group of models, specified by an array of file names.

Both these routines are applicable to models of one of these types: LP, QP, QCP, and MIP. (Neither applies to networks.) Advanced bases in `.sav` files are ignored by these routines. Acceptable file formats for these routines are:

- ◆ `.mps`
- ◆ `.lp`
- ◆ `.sav`

Parameter settings in the environment control the resources for the tuning tool; the parameter settings passed in the arguments of these routines are used by ILOG CPLEX as a starting point for tuning. You can specify parameters and corresponding values for ILOG CPLEX to use as a starting point. Bear in mind that the parameters you specify at nondefault settings will be fixed during the tuning. That is, the tuning tool will not attempt to modify them. In other words, when you set a parameter as a starting point, you eliminate it from consideration by the tuning tool.

Callbacks (except the tuning callback) are ignored by these tuning routines in the Callable Library (C API).

The tuning tool checks the termination signal (that is, the variable set by the routine `CPXsetterminate`) and terminates the tuning process in compliance with that termination signal. Tuning also respects time limits set in the environment.

The result of either tuning routine is an environment containing the parameter settings recommended by tuning. You can query these values or write them to a formatted PRM file with the usual routines.

After tuning, the return value of either routine specifies 0 (zero) when tuning completed successfully and nonzero when it has not done so. A tuning status is also returned as an argument to these routines. The possible nonzero values of the tuning status argument are these:

- ◆ `CPX_TUNE_ABORT` specifies that abort occurred through `CPXsetterminate`.
- ◆ `CPX_TUNE_TILIM` specifies that tuning reached a time limit specified as a parameter in the environment.

Tuning will set any parameters which have been chosen even if there is an early termination.

Callbacks for tuning

A tuning callback is a user-written function that ILOG CPLEX calls before each trial run during a tuning session. A tuning callback allows you to follow the progress of tuning. It reports information that enables you to estimate how much more time tuning needs to achieve results useful in your model.

To use a tuning callback in a tuning session, you must first write the callback function, and then pass it to ILOG CPLEX. ILOG CPLEX will then execute your tuning callback before it begins a trial run.

- ◆ In Concert Technology, you must implement your user-written function as an instance of the tuning callback class.
 - `IloCplex::TuningCallbackI` in the C++ API
 - `IloCplex.TuningCallback` in the Java API
 - `Cplex.TuningCallback` in the .NET API

For details about writing your tuning callback, see *Using optimization callbacks*, especially *Implementing callbacks in ILOG CPLEX with Concert Technology*

- ◆ In the Callable Library (C API), use the routine `CPXset tuningcallbackfunc`. For more about how to write a callback, see also *Implementing callbacks in the Callable Library*.

Terminating a tuning session

To terminate a tuning session, you can use one of the following means:

- ◆ In the C++ API, pass an instance of the class `IloCplex::Aborter` to an instance of `IloCplex`. Then call the method `IloCplex::Aborter::abort` to terminate the tuning session.
- ◆ In the Java API, pass an instance of the class `IloCplex.Aborter` to an instance of `IloCplex`. Then call the method `IloCplex.Aborter.abort` to terminate the tuning session.
- ◆ In the .NET API, pass an instance of the class `Cplex.Aborter` to an instance of `Cplex`. Then call the method `Cplex.Aborter.Abort` to terminate the tuning session.
- ◆ In the Callable Library (C API), call the routine `CPXsetterminate` to set a pointer to the termination signal. Initially, the value of the termination signal should be zero. When your application sets the termination signal to a nonzero value, then ILOG CPLEX will terminate the tuning session.

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.

In this section

Solving LPs: simplex optimizers

Documents the primal and dual simplex optimizers.

Solving LPs: barrier optimizer

Documents solving linear programming problems by means of the ILOG CPLEX Barrier Optimizer.

Solving network-flow problems

Documents the ILOG CPLEX Network Optimizer.

Solving problems with a quadratic objective (QP)

Describes solving convex quadratic programming problems (QPs) with ILOG CPLEX.

Solving problems with quadratic constraints (QCP)

Documents the solution of *quadratically constrained* programming problems (QCPs), including the special case of *second order cone* programming problems (SOCPs).

Solving LPs: simplex optimizers

Documents the primal and dual simplex optimizers.

In this section

Introducing the primal and dual optimizers

Places the primal and dual optimizers in context.

Choosing an optimizer for your LP problem

Explains considerations of choosing an optimizer for LP models.

Tuning LP performance

Documents tactics for tuning performance on LP models.

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.

Diagnosing LP infeasibility

Documents ways to diagnose sources of infeasibility in LP models.

Examples: using a starting basis in LP optimization

Shows examples of starting from an advanced basis for LP optimization.

Introducing the primal and dual optimizers

The preceding topics 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 topic and the following one tell you more about solving linear programs with the LP optimizers of ILOG CPLEX. This topic emphasizes primal and dual simplex optimizers.

Choosing an optimizer for your LP problem

Explains considerations of choosing an optimizer for LP models.

In this section

Overview of LP optimizers

Introduces parameters to select LP optimizers.

Automatic selection of an optimizer

Describes conditions for automatic selection of an optimizer.

Dual simplex optimizer

Describes conditions favoring the dual simplex optimizer.

Primal simplex optimizer

Describes conditions favoring the primal simplex optimizer.

Network optimizer

Describes conditions favoring the network optimizer.

Barrier optimizer

Describes conditions favoring the barrier optimizer.

Sifting optimizer

Describes conditions favoring the sifting optimizer.

Concurrent optimizer

Describes conditions favoring the concurrent optimizer.

Parameter settings and optimizer choice

Describes special considerations about parameters and choice of optimizer.

Overview of LP optimizers

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 Technology or the Callable Library, or you use the Interactive Optimizer.

The choice of LP optimizer in ILOG CPLEX can be specified using the *algorithm for continuous problems* parameter, named `RootAlg` in the C++, Java, and .NET APIs, `CPX_PARAM_LPMETHOD` in the Callable Library, and `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 sets which optimizer will be used when you solve a model in one of the following ways:

- ◆ `plex.solve` (Concert Technology)
- ◆ `CPXlpopt` (Callable Library)
- ◆ `optimize` (Interactive Optimizer)

The choices for `LPMethod` are summarized in *Settings of the LPMethod parameter for choosing an optimizer*.

Settings of the LPMethod parameter for choosing an optimizer

Setting of LPMethod	Meaning	See Section
0	Default Setting	<i>Automatic selection of an optimizer</i>
1	Primal Simplex	<i>Primal simplex optimizer</i>
2	Dual Simplex	<i>Dual simplex optimizer</i>
3	Network Simplex	<i>Network optimizer</i>
4	Barrier	<i>Barrier optimizer</i>
5	Sifting	<i>Sifting optimizer</i>
6	Concurrent Dual, Barrier, and Primal	<i>Concurrent optimizer</i>

The symbolic names for these settings in an application are summarized in *Symbolic names for LP solution methods*.

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 an optimizer

The default `Automatic` setting of the LP method lets ILOG CPLEX decide which 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 dual simplex optimizer being called if the parallel mode is deterministic, and in the concurrent optimizer being called if the parallel mode is opportunistic. An exception to this rule occurs 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* 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*.

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*, especially *Concurrent optimizer*.

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

Documents tactics for tuning performance on LP models.

In this section

Introducing performance tuning for LP models

Outlines general facilities for performance tuning on LP models.

Preprocessing

Documents preprocessing at default parameter settings in LP optimizers.

Starting from an advanced basis

Documents effect of an advanced basis on LP optimizers.

Simplex parameters

Documents parameters settings that may improve performance of LP optimizers.

Introducing performance tuning for LP models

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*.

To help you decide whether default settings of parameters are best for your model, or whether other parameter settings may improve performance, the tuning tool is available. *Tuning tool* explains more about this utility and offers you examples of its use.

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

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. *Dependency checking parameter DepInd or CPX_PARAM_DEPIND* shows you the possible settings of the *dependency switch*, the parameter that controls dependency checking, and indicates their effects.

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 memory emphasis parameter (*memory reduction switch*: `MemoryEmphasis (bool)`, `CPX_PARAM_MEMORYEMPHASIS (int)`) 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 memory emphasis parameter

(*memory reduction switch*: `MemoryEmphasis` (bool), `CPX_PARAM_MEMORYEMPHASIS` (int)).

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 decide 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 the 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` (zero).

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 read an advanced basis from this file later:

- ◆ 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`.

Tip: A basis file, also known as a BAS file, is a formatted text file conforming to the MPS standard. It relies on each variable (column) and each constraint (row) having a name. If those names do not exist, names will be created automatically and added during write operations of a basis file. If you anticipate the need to read and write basis files, it is a good idea to assign a name yourself to every variable and constraint when you create your model.

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*). The simplex tuning suggestions appear in the following topics:

- ◆ *Pricing algorithm and gradient parameters*
- ◆ *Scaling*
- ◆ *Crash*
- ◆ *Memory management and problem growth*

Pricing algorithm and gradient parameters

The parameters in *DPriInd parameter settings for dual simplex pricing algorithm* set 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 `plex.log`. *Managing log files* 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 sets 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 *DPriInd parameter settings for dual simplex pricing algorithm* and *PPriInd parameter settings for primal simplex pricing algorithm*. The default value in both cases is 0 (zero).

DPriInd parameter settings for dual simplex pricing algorithm

	Description	Concert	Callable Library
0	set 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

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.

Use the scaling parameter (*scale parameter*: `ScaInd`, `CPX_PARAM_SCAIND`) to set scaling appropriate for your model. *Scalnd parameter settings for scaling methods* summarizes available values for this parameter.

Scalnd parameter settings for scaling methods

Scalnd Value	Meaning
-1	no scaling
0	equilibration scaling (default)
1	aggressive scaling

Refactoring frequency

ILOG CPLEX dynamically decides 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` parameter sets 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 *CraInd parameter settings for the dual simplex optimizer*.

CraInd parameter settings for the dual simplex optimizer

CraInd Setting	Meaning for Dual Simplex Optimizer
1	Use default starting basis guided by coefficients
0	Use an aggressive starting basis ignoring coefficients
-1	Use an aggressive starting basis contrary to coefficients

In the primal simplex optimizer, the `CraInd` setting sets 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 *CraInd parameter settings for the primal simplex optimizer*.

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.

In this section

Lack of memory

Documents ILOG CPLEX behavior in limited memory for LP models.

Numeric difficulties

Documents ILOG CPLEX behavior with respect to numeric difficulties in LP models.

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 decide 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 (*memory reduction switch*) can help you in this respect.

- ◆ C++ Name `MemoryEmphasis` (`bool`) in Concert Technology
- ◆ C Name `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 set 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 precision emphasis* parameter controls the degree of numerical caution used during optimization of a model.

- ◆ C++ Name `NumericalEmphasis` in Concert Technology
- ◆ C Name `CPX_PARAM_NUMERICALEMPHASIS` 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 trade-off 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 $1/3$ as $.33333333$ on a computer that in principle computes values to about 16 digits can introduce an apparent “exact” value that will be treated as given but may not represent what you intend. This difficulty is compounded if similar or related values are represented a little differently elsewhere in the model.

For example, consider the constraint $1/3 x_1 + 2/3 x_2 = 1$. 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 + 0.6666667 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: 0.333333333 x1 + 0.666666667 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 c_1 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 *messages to screen switch* `CPX_PARAM_SCRIND` is set to 1 (one)). After 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* 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* 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. After 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* perturbation constant (`EpPer` 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 `EpMrk` is 0.01, and suitable values range from 0.0001 to 0.99999.

Sometimes slow progress in Phase I (the period when ILOG CPLEX calculates the first feasible solution) is due to similar 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

Documents ways to diagnose sources of infeasibility in LP models.

In this section

Infeasibility reported by LP optimizers

Documents facilities for diagnosing sources of infeasibility in LP models.

Coping with an ill-conditioned problem or handling unscaled infeasibilities

Describes the effect of scaling on infeasible LP models.

Interpreting solution quality

Documents facilities for evaluating solution quality in LP models.

Finding a conflict

Introduces the conflict refiner as a tool for diagnosing sources of infeasibility in a LP models.

Repairing infeasibility: FeasOpt

Introduces FeasOpt as a tool for repairing infeasibility in LP models.

Infeasibility reported by LP optimizers

ILOG CPLEX reports statistics about any problem that it optimizes. For infeasible solutions, it reports values that you can analyze to discover 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 in which the detection of infeasibility occurred 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*.
- ◆ 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*.

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 decide 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*, or after you reduce feasibility tolerance, as explained in *Maximum bound infeasibility: identifying largest bound violation*. 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*. 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 detected 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 see whether ILOG CPLEX has also concluded that the model has a feasible solution, use one of these routines or methods:

- ◆ in Concert Technology:
 - `isPrimalFeasible` or `isDualFeasible` in the C++ API;
 - `IloCplex.isPrimalFeasible` or `isDualFeasible` in the Java API;
 - `Cplex.IsPrimalFeasible` or `IsDualFeasible` in the .NET API.
- ◆ `CPXsolninfo` in the Callable Library.

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*, and for Java applications, see *Accessing solution information*. 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 detects that 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 detects that 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 discover the cause of infeasibility in your problem. If the largest bound violation exceeds the feasibility tolerance of your problem by only a small amount, then you may be able to get a feasible solution to the problem by increasing the *feasibility tolerance* parameter (E_pRHS in Concert Technology, CPX_PARAM_EPRHS in the Callable Library). Its range is between $1e-9$ and 0.1 . Its default value is $1e-6$.

Maximum reduced-cost infeasibility

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

To change the optimality tolerance, set the *optimality tolerance* parameter (E_pOpt 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 decide 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 decide 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 `getQuality` or the routine `CPXgetdblquality` to access the information provided by the command `display solution quality` in the Interactive Optimizer.

If you discover 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* 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*.

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*

Examples: using a starting basis in LP optimization

Shows examples of starting from an advanced basis for LP optimization.

In this section

Overview

Introduces an example starting from an advanced basis.

Example `ilolpex6.cpp`

Shows an example of starting from in advanced basis in the C++ API.

Example `lpex6.c`

Shows an example of starting from an advanced basis in the C API.

Overview

Here is an approach mentioned in the section *Tuning LP performance*, which 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* is from Concert Technology in the C++ API. *Example lpex6.c* 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 detected as optimal, 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

Documents solving linear programming problems by means of the ILOG CPLEX Barrier Optimizer.

In this section

Introducing the barrier optimizer

Describes the type of problem the barrier optimizer solves and characterizes its approach.

Barrier simplex crossover

Describes considerations about basis solutions and the barrier optimizer.

Differences between barrier and simplex optimizers

Contrasts barrier optimizer with simplex optimizers.

Using the barrier optimizer

Documents parameters and methods associated with the barrier optimizer.

Special options in the Interactive Optimizer

Describes where to find further options of the barrier optimizer in the Interactive Optimizer.

Controlling crossover

Documents values of the parameter controlling barrier crossover.

Using SOL file format

Describes content of SOL file with respect to the barrier optimizer and crossover.

Interpreting the barrier log file

Describes the log file generated by the barrier optimizer.

Understanding solution quality from the barrier LP optimizer

Documents the information available about the quality of a solution found by the barrier optimizer.

Tuning barrier optimizer performance

Describes facilities for tuning performance of the barrier optimizer.

Overcoming numeric difficulties

Documents ways to cope with numeric difficulties in the barrier optimizer.

Diagnosing infeasibility reported by barrier optimizer

Explains information about an infeasible solution from the barrier optimizer and its implications in diagnosing sources of infeasibility.

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:

$$\text{Minimize } c^T x$$

$$\text{subject to } Ax = b$$

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

where 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:

$$\text{Maximize } b^T y - u^T w + l^T z$$

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

$$\text{with these bounds } w \geq 0 \text{ and } z \geq 0$$

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 $(Ax = b, x + s = u)$ and dual constraints $(A^T y + z - w = c)$ 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 AA^T and then computes a Cholesky factor of it. This factored matrix has the same number of

nonzeros on each iteration. The number of nonzeros in this matrix is influenced by the barrier ordering parameter.

The 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*, and *Nonzeros in lower triangle of AAT in the log file*).

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* 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 uses 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*, 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*, may be helpful.

Using the barrier optimizer

As you have read in *Introducing the barrier optimizer*, 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 *Settings of LPMETHOD to invoke the barrier optimizer*.

Settings of LPMETHOD to 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 *Calling the barrier optimizer*.

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 the Interactive Optimizer

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 *BarCrossAlg parameter settings*.

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*, for more about solution files.

Interpreting the barrier log file

Describes the log file generated by the barrier optimizer.

In this section

Accessing and managing the log file of the barrier optimizer

Describes routines and methods to access and manage the log file of the barrier optimizer.

Sample log file from the barrier optimizer

Shows a typical log file from the barrier optimizer.

Preprocessing in the log file

Identifies the preprocessing report in a log file from the barrier optimizer.

Nonzeros in lower triangle of AAT in the log file

Explains report of nonzeros in the log file of the barrier optimizer.

Ordering-algorithm time in the log file

Describes the report of the time used by ordering algorithm of the barrier optimizer.

Cholesky factor in the log file

Identifies the report of the Cholesky factor in the log file of the barrier optimizer.

Iteration progress in the log file

Identifies iteration progress in the log file of the barrier optimizer.

Infeasibility ratio in the log file

Identifies the infeasibility ratio reported in the log file of the barrier optimizer.

Accessing and managing the log file of the barrier optimizer

Like the ILOG CPLEX simplex optimizers, the barrier optimizer records information about its progress in a log file as it works. Some users find it helpful to keep a new log file for each session. By default, ILOG CPLEX records information in a file named `cplex.log`.

- ◆ In Concert Technology, use the method `setOut` of `IloCplex` to designate a logfile for a predefined channel.
- ◆ In the Callable Library, use the routine `CPXsetlogfile` with arguments to indicate the log file.
- ◆ In the Interactive Optimizer, use the command `set logfile filename` to change the name of 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 *BarDisplay parameter settings*.

BarDisplay parameter settings

BarDisplay Values	Meaning
0	no display
1	display normal information (default)
2	display detailed (diagnostic) output

Sample log file from the barrier optimizer

Here is an example of a log file for a barrier optimization (without crossover):

```
Tried aggregator 1 time.
LP Presolve eliminated 9 rows and 11 columns.
Aggregator did 6 substitutions.
Reduced LP has 12 rows, 15 columns, and 38 nonzeros.
Presolve time = 0.00 sec.
Number of nonzeros in lower triangle of A*A' = 26
Using Approximate Minimum Degree ordering
Total time for automatic ordering = 0.00 sec.
Summary statistics for Cholesky factor:
  Rows in Factor          = 12
  Integer space required  = 12
  Total non-zeros in factor = 78
  Total FP ops to factor  = 650
Itn   Primal Obj      Dual Obj  Prim Inf  Upper Inf  Dual Inf
 0  -1.3177911e+01  -1.2600000e+03  6.55e+02  0.00e+00  3.92e+01
 1  -4.8683118e+01  -5.4058675e+02  3.91e+01  0.00e+00  1.18e+01
 2  -1.6008142e+02  -3.5969226e+02  1.35e-13  7.11e-15  5.81e+00
 3  -3.5186681e+02  -6.1738305e+02  1.59e-10  1.78e-15  5.16e-01
 4  -4.5808732e+02  -4.7450513e+02  5.08e-12  1.95e-14  4.62e-02
 5  -4.6435693e+02  -4.6531819e+02  1.66e-12  1.27e-14  1.59e-03
 6  -4.6473085e+02  -4.6476678e+02  5.53e-11  2.17e-14  2.43e-15
 7  -4.6475237e+02  -4.6475361e+02  5.59e-13  2.99e-14  2.19e-15
 8  -4.6475312e+02  -4.6475316e+02  1.73e-13  1.55e-14  1.17e-15
 9  -4.6475314e+02  -4.6475314e+02  1.45e-13  2.81e-14  2.17e-15

Barrier - Optimal: Objective = -4.6475314194e+02
Solution time = 0.01 sec. Iterations = 9
```

Preprocessing in the log file

The opening lines of that log file record information about preprocessing by the ILOG CPLEX presolver and aggregator. After those preprocessing statistics, the next line records the number of nonzeros in the lower triangle of a particular matrix, 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*.) 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 $ATy + z - w = c$. As a result, in the log file, the infeasibilities represent norms, as summarized in *Infeasibilities and norms in the log file of a barrier optimization*.

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 `plex.getQuality` or use the routines `CPXgetintquality` for integer information and `CPXgetdblquality` for double-valued information.

Barrier solution quality display lists the items ILOG CPLEX displays and explains their meaning. In the solution quality display, the term `pi` refers to dual solution values, that is, the y values in the conventional barrier problem-formulation. The term `rc` refers to reduced cost, that is, the difference $z - w$ in the conventional barrier problem-formulation. Other terms are best understood in the context of primal and dual LP formulations.

Normalized errors, for example, represent the accuracy of satisfying the constraints while considering the quantities used to compute 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.

Barrier solution quality display

Item	Meaning
primal objective	primal objective value $c^T x$
dual objective	dual objective value $b^T y - u^T w + l^T z$

Item	Meaning
duality gap	difference between primal and dual objectives
complementarity	sum of column and row complementarity
column complementarity (total)	sum of $ (x_j - l_j) \cdot z_j + (u_j - x_j) \cdot w_j $
column complementarity (max)	maximum of $ (x_j - l_j) \cdot z_j $ and $ (u_j - x_j) \cdot w_j $ over all variables
row complementarity (total)	sum of $ \text{slack}_i \cdot y_i $
row complementarity (max)	maximum of $ \text{slack}_i \cdot 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 ($A^T p_i + rc = c$) (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 ($A^T p_i + rc = c$) (max)	accuracy of dual constraints

Tuning barrier optimizer performance

Describes facilities for tuning performance of the barrier optimizer.

In this section

Overview of parameters for tuning the barrier optimizer

Introduces performance parameters of the barrier optimizer.

Memory emphasis: letting the optimizer use disk for storage

Describes effect of the memory emphasis parameter on the performance of the barrier optimizer.

Preprocessing

Describes the effect of preprocessing on the performance of the barrier optimizer.

Detecting and eliminating dense columns

Explains the effect of dense columns in the model on performance of the barrier optimizer.

Choosing an ordering algorithm

Describes performance considerations with respect to the ordering algorithm of the barrier optimizer.

Using a starting-point heuristic

Describes impact of starting-point heuristics on performance of the barrier algorithm.

Overview of parameters for tuning the barrier optimizer

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.

To help you decide whether default settings of parameters are best for your model, or whether other parameter settings may improve performance, the tuning tool is available. *Tuning tool* explains more about this utility and shows you examples of its use.

Parameters for performance and numeric difficulties

The following sections document parameters particularly relevant to performance and numeric difficulties in the barrier optimizer:

- ◆ *Memory emphasis: letting the optimizer use disk for storage*
- ◆ *Preprocessing;*
- ◆ *Detecting and eliminating dense columns;*
- ◆ *Choosing an ordering algorithm;*
- ◆ *Using a starting-point heuristic.*

Parameters for termination

In addition, several parameters set termination criteria. With them, you control when ILOG CPLEX stops optimization.

Parameters to control convergence

You can also control convergence tolerance (another factor that influences performance). Convergence tolerance specifies 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.

Cholesky factor and parameter changes

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*.

Dense columns and performance considerations

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 AAT in the log file*. *Detecting and eliminating dense columns* also says more about column density.

Managing parameter changes for the barrier optimizer

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.). After you have found satisfactory parametric settings, keep them in a parameter specification file for re-use, as explained in *Saving a parameter specification file* 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 (*memory reduction switch*) to 1 (one) instead of its default value of 0 (zero).

- ◆ `MemoryEmphasis` in Concert Technology
- ◆ `CPX_PARAM_MEMORYEMPHASIS` 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 (*memory available for working storage*).

- ◆ `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 (*directory for working files*).

- ◆ `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. *Dependency checking parameter* *DepInd* or *CPX_PARAM_DEPIND* shows you the possible settings of the *dependency switch*, the parameter that controls dependency checking, and indicates their effects.

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 AAT in the log file 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 (zero) means that ILOG CPLEX will set 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*, 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 specified by the `BarStartAlg` parameter, and *BarStartAlg parameter settings for starting-point heuristics* summarizes the possible settings and their meanings.

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 *barrier starting point algorithm* parameter `IloCplex::BarStartAlg` or `CPX_PARAM_BARSTARTALG`.

Overcoming numeric difficulties

Documents ways to cope with numeric difficulties in the barrier optimizer.

In this section

Default behavior of the barrier optimizer with respect to numeric difficulty

Describes the context of crossover by barrier optimizer.

Numerical emphasis settings

Documents purpose of numerical emphasis parameter for barrier optimizer.

Difficulties in the quality of solution

Suggests strategies with the barrier optimizer to overcome poor quality solutions.

Difficulties during optimization

Suggests strategies for overcoming difficulties during optimization with the barrier optimizer.

Difficulties with unbounded problems

Suggests strategies to overcome unbounded problems with the barrier optimizer.

Default behavior of the barrier optimizer with respect to numeric difficulty

As noted in *Differences between barrier and simplex optimizers*, 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 precision emphasis* parameter.

- ◆ `NumericalEmphasis` in Concert Technology
- ◆ `CPX_PARAM_NUMERICAL EMPHASIS` 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 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* and recapitulated here in the following subsections.

Change the barrier algorithm

The ILOG CPLEX Barrier Optimizer implements the algorithms listed in *BarAlg parameter settings for barrier optimizer*. 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.

BarAlg parameter settings for barrier optimizer

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*, 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*, 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 *BarStartAlg parameter settings for starting-point heuristics*.

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 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 decide 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 detect 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* and *Diagnosing infeasibility by refining conflicts* 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 find 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 discovers 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

Documents the ILOG CPLEX Network Optimizer.

In this section

Choosing an optimizer: network considerations

Describes conditions under which the network optimizer is appropriate.

Formulating a network problem

Defines the characteristics of a network flow model.

Example: network optimizer in the Interactive Optimizer

Demonstrates an example of the network optimizer in the Interactive Optimizer.

Solving problems with the network optimizer

Explains invocation and activity of the network optimizer.

Example: using the network optimizer with the Callable Library netex1.c

Demonstrates an example of the network optimizer in the C API.

Solving network-flow problems as LP problems

Explains the conversion between a network flow model and a conventional LP model.

Example: network to LP transformation netex2.c

Demonstrates an example of converting a network flow model to its LP model in the C API.

Choosing an optimizer: network considerations

As explained in *Using the Callable Library in an application*, 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, sets the minimum flow allowed through the arc a . By default, the lower bound on an arc is 0 (zero).
- ◆ u_a , the upper bound, sets the maximum flow allowed through the arc a . By default, the upper bound on an arc is positive infinity.
- ◆ c_a , the objective value, specifies the contribution to the objective function of one unit of flow through the arc.

Each node n is associated with one value:

- ◆ s_n is the supply value at node n .

By convention, a node with strictly positive supply value (that is, $s_n > 0$) is called a *supply* node or a *source*, and a node with strictly negative supply value (that is, $s_n < 0$) is called a *demand* node or a *sink*. A node where $s_n = 0$ is called a *transshipment* node. The sum of all supplies must match the sum of all demands; if not, then the network flow problem is *infeasible*.

T_n is the set of arcs whose tails are node n ; H_n is the set of arcs whose heads are node n . The usual form of a network problem looks like this:

Minimize (or maximize)

$$\sum_{a \in A} (c_a x_a)$$

subject to

$$\sum_{\alpha \in T_n} x_\alpha - \sum_{\alpha \in H_n} x_\alpha = s_n \quad (\forall n \in N)$$

with these bounds

$$l_\alpha \leq x_\alpha \leq u_\alpha \quad (\forall \alpha \in A)$$

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

Demonstrates an example of the network optimizer in the Interactive Optimizer.

In this section

Network flow problem description

Describes a sample network flow problem for the network optimizer.

Understanding the network log file

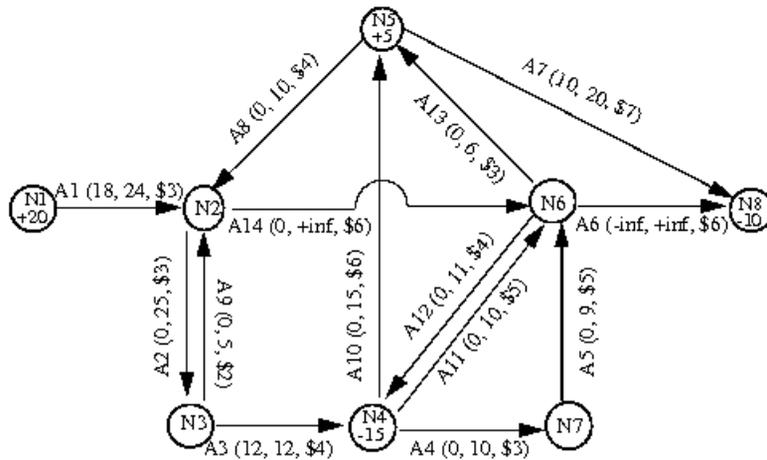
Describes a typical log file, corresponding to the example, from the network optimizer.

Tuning performance of the network optimizer

Suggests strategies for improving performance of the network optimizer.

Network flow problem description

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. A *directed network with arc-capacity, flow-cost, sinks, and sources* 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.



A directed network with arc-capacity, flow-cost, sinks, and sources

The example in *A directed network with arc-capacity, flow-cost, sinks, and sources* 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 = 2.6900000000e+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

Explains invocation and activity of the network optimizer.

In this section

Invoking the network optimizer

Describes invocation of the network optimizer.

Network extraction

Documents how the network optimizer extracts a problem.

Preprocessing and the network optimizer

Explains possible impact of the preprocessor on network flow models.

Invoking the network optimizer

You instruct ILOG CPLEX to apply the network optimizer for solving the LP at hand by setting the *algorithm for continuous problems* parameter:

- ◆ setting CPX_PARAM_LPMETHOD to CPX_ALG_NET in the Callable Library
- ◆ or setting RootAlg 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 *algorithm for continuous quadratic optimization* parameter CPX_PARAM_QPMETHOD to CPX_ALG_NET. For Concert Technology, you set the RootAlg parameter 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 qualify 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 *A directed network with arc-capacity, flow-cost, sinks, and sources*.

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*, `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 *messages to screen switch* 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*.

$$0 \leq a_{13} \leq 6 \quad 0 \leq a_{14}$$

In that formulation, in each column there is exactly one coefficient equal to 1 (one), exactly one coefficient equal to -1, and all other coefficients are 0 (zero).

Since a network-flow problem corresponds in this way to an LP problem, you can indeed solve a network-flow problem by means of a ILOG CPLEX LP optimizer as well. If you read a network-flow problem into the Interactive Optimizer, you can transform it into its LP formulation with the command `change problem lp`. After this change, you can apply any of the LP optimizers to this problem.

When you change a network-flow problem into an LP problem, the basis information that is available in the network-flow problem is passed along to the LP formulation. In fact, if you have already solved the network-flow problem to optimality, then if you call the primal or dual simplex optimizers (for example, with the Interactive Optimizer command `primopt` or `tranopt`), that simplex optimizer will perform no iterations.

Generally, you can also use the same basis from a basis file for both the LP and the network optimizers. However, there is one exception: in order to use an LP basis with the network optimizer, at least one slack variable or one artificial variable needs to be basic. *Starting from an advanced basis* explains more about this topic in the context of LP optimizers.

If you have already read the LP formulation of a problem into the Interactive Optimizer, you can transform it into a network with the command `change problem network`. Given any LP problem and this command, ILOG CPLEX will try to find the largest network embedded in the LP problem and transform it into a network-flow problem. However, as it does so, it discards all rows and columns that are not part of the embedded network. At the same time, ILOG CPLEX passes along as much basis information as possible to the network optimizer.

Example: network to LP transformation `netex2.c`

This example shows how to transform a network-flow problem into its corresponding LP formulation. That example also indicates why you might want to make such a change. The example reads a network-flow problem from a file (rather than populating the problem object by adding rows and columns as in `netex1.c`). 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 detects 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)

Describes solving convex quadratic programming problems (QPs) with ILOG CPLEX.

In this section

Identifying convex QPs

Explains how to identify a convex quadratic program.

Entering QPs

Documents the two views of quadratic objective functions supported by ILOG CPLEX: a matrix view and an algebraic view.

Saving QP problems

Recommends appropriate file formats to save a quadratic program.

Changing problem type in QPs

Explains special considerations about the change of the problem type of a quadratic program.

Changing quadratic terms

Defines quadratic algebraic term and quadratic matrix.

Optimizing QPs

Describes how to invoke an optimizer for a quadratic program and explains the appropriate choice of optimizer.

Diagnosing QP infeasibility

Explains infeasibility in the context of a quadratic program.

Examples: creating a QP, optimizing, finding a solution

Demonstrates creation, optimization, and solution of a quadratic program.

Example: reading a QP from a file qpex2.c

Demonstrates reading data for a quadratic program from a file and solving in the C API.

Identifying convex QPs

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

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

subject to $Ax \sim b$

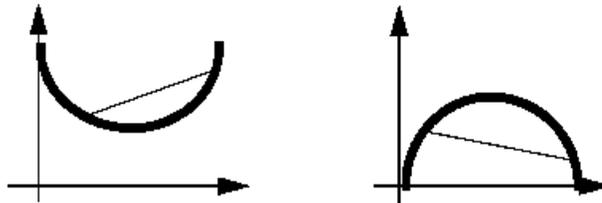
with these bounds $l \leq x \leq u$

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. Q is a matrix of objective function coefficients. That is, the elements Q_{ij} are the coefficients of the quadratic terms x_i^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 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. *Minimize a convex objective function, maximize a concave objective function* 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.



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 Q is positive semi-definite or negative semi-definite. For convex QPs, Q must be positive semi-definite; that is, $x^T Q x \geq 0$ for every vector x , whether or not x is feasible. For concave maximization problems, the requirement is that Q must be negative semi-definite; that is, $x^T Q x \leq 0$ for every vector 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 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 Q are of the correct sign. For a nonseparable case, it may be less easy to decide in advance the convexity of Q . However, ILOG CPLEX detects 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* of the preface of this manual, may be helpful.

Entering QPs

Documents the two views of quadratic objective functions supported by ILOG CPLEX: a matrix view and an algebraic view.

In this section

Matrix view

Describes the matrix view of a quadratic program.

Algebraic view

Describes the algebraic view of a quadratic program.

Examples for entering QPs

Demonstrates ways to enter the objective function of a quadratic program.

Reformulating QPs to save memory

Describes an alternative formulation of a quadratic program that may save memory.

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 explicit when you enter 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*, or provide as input $(\mathbf{Q} + \mathbf{Q}')/2$ instead of \mathbf{Q} . This latter approach relies on the identity $\mathbf{Q} = (\mathbf{Q} + \mathbf{Q}')/2 + (\mathbf{Q} - \mathbf{Q}')/2$ combined with the fact that $(\mathbf{Q} - \mathbf{Q}')/2$ 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 you enter a quadratic objective function 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 you enter 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 explicitly specified in the file.

```
Minimize
obj: [ 100 x1 ^2 - 200 x1 * x2 + 100 x2 ^2 ] / 2
```

MPS format for this same objective function contains 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 may choose to simplify that 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 is 200, not 400.

Reformulating QPs to save memory

When the \mathbf{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 \mathbf{Q} as $\mathbf{F}\mathbf{F}'$, (where \mathbf{F} is another matrix, not necessarily square, having fewer nonzeros than \mathbf{Q} , and \mathbf{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 \mathbf{F} .

Portfolio optimization models in particular benefit from this reformulation. In the most common portfolio models, \mathbf{Q} is a covariance matrix of asset returns, while \mathbf{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 \mathbf{F} corresponds to the number of time periods for which returns are measured.

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

Furthermore, if \mathbf{F} is a sparser matrix than \mathbf{Q} , this alternative formulation may improve performance even if \mathbf{F} has more columns than \mathbf{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. (These formats are documented in *ILOG CPLEX File Formats Reference Manual*.) 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 a linear program (LP) from a QP. The following sections discuss the topic for these users.

When you enter a problem, ILOG CPLEX discovers 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)*.) 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)*.

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 must also 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` for the LP case or `CPXPROB_QP` for the QP case 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:

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

$$\text{subject to } a + b \geq 10$$

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, for example, 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 in the **Interactive Optimizer**, 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 $Q = 0$.

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 Technology** applications, use the method `solve`.

With default settings, this will result in ILOG CPLEX invoking the barrier optimizer 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 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 root algorithm parameter (`QPMETHOD` in the **Interactive Optimizer** and in the **Callable Library**). *RootAlg parameter settings for QPs* shows you the possible settings.

RootAlg parameter settings for QPs

Root algorithm 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. In case your model is not solved satisfactorily by default settings, consider the advice offered in *Solving LPs: barrier optimizer*, especially *Tuning barrier optimizer performance* as well as in *Solving LPs: simplex optimizers*, especially *Tuning LP performance*.

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*.

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*) 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.

Examples: creating a QP, optimizing, finding a solution

Demonstrates creation, optimization, and solution of a quadratic program.

In this section

Problem description of a quadratic program

Describes the model used in the examples of a quadratic program.

Example: iloqpex1.cpp

Demonstrates the solution of a quadratic program in the C++ API.

Example: QPex1.java

Demonstrates the solution of a quadratic program in the Java API.

Example: qpex1.c

Demonstrates the solution of a quadratic program in the C API.

Problem description of a quadratic program

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

$$0 \leq x_1 \leq 40$$

$$0 \leq x_2 \leq +\infty$$

$$0 \leq x_3 \leq +\infty$$

The following sections solve this model in the various APIs available in ILOG CPLEX:

- ◆ *Example: `iloqpex1.cpp`*
- ◆ *Example: `QPex1.java`*
- ◆ *Example: `qpex1.c`*

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 a QP objective function. 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*, where automatic transformation with logical constraints can render a problem MIQP.)

The complete program `iloqpex1.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. 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, `iloqpex1.cpp` and `QPex1.java`, are included online in the standard distribution.

Instead of calling `CPXlpopt` to find a solution as for the *linear* programming problem in `lpex1.c`, this example calls `CPXqpopt` to optimize this *quadratic* programming problem.

Like other applications based on the ILOG CPLEX Callable Library, this one begins with calls to `CPXopenCPLEX` to initialize the ILOG CPLEX environment and to `CPXcreateprob` to create the problem object. Before it ends, it frees the problem object with a call to `CPXfreeprob`, and it frees the environment with a call to `CPXcloseCPLEX`.

In the routine `setproblemdata`, there are parameters for `qmatbeg`, `qmatcnt`, `qmatind`, and `qmatval` to fill the quadratic coefficient matrix. The Callable Library routine `CPXcopyquad` copies this data into the problem object created by the Callable Library routine `CPXcreateprob`.

In this example, the problem is a maximization, so the objective sense is specified as `CPX_MAX`.

The off-diagonal terms in the matrix 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 `lpex2.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)**

Documents the solution of *quadratically constrained* programming problems (QCPs), including the special case of *second order cone* programming problems (SOCPs).

In this section

Identifying a quadratically constrained program (QCP)

Defines the types of quadratically constrained programs that ILOG CPLEX solves.

Detecting the problem type of a QCP or SOCP

Documents the criteria that ILOG CPLEX components use to detect the problem type of a quadratically constrained program.

Changing problem type

Explains considerations about changing the problem type in a quadratically constrained program, according to ILOG CPLEX components.

Changing quadratic constraints

Explains special considerations about modifying a constraint containing a quadratic term.

Solving with quadratic constraints

Documents the routine or method to solve a quadratically constrained program.

Numeric difficulties and quadratic constraints

Describes the symptoms of numeric difficulties in a quadratically constrained program.

Examples: QCP

Tells where to find sample applications solving a quadratically constrained program.

Identifying a quadratically constrained program (QCP)

Defines the types of quadratically constrained programs that ILOG CPLEX solves.

In this section

Characteristics of a quadratically constrained program

Describes the characteristics of a quadratically constrained program.

Convexity

Defines convexity in the context of a quadratically constrained program.

Semi-definiteness

Defines semi-definiteness in the context of a quadratically constrained program.

Second order cone programming (SOCP)

Relates convexity and positive semi-definiteness to second order cone programming.

Characteristics of a quadratically constrained program

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

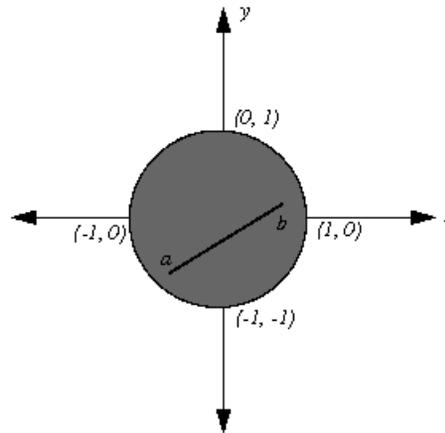
$$\text{and } \mathbf{a}_i^T\mathbf{x} + \mathbf{x}^T\mathbf{Q}_i\mathbf{x} \leq r_i \text{ for } i=1,\dots,q$$

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

As in 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 \mathbf{Q}_i matrix be positive semi-definite (PSD) or that the constraint can be transformed into a *second order cone*. The following sections offer more information about these concepts.

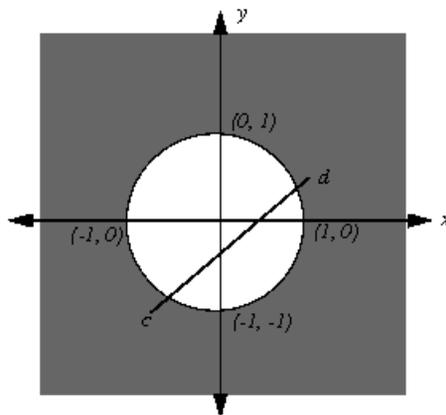
Convexity

The inequality $x^2 + y^2 \leq 1$ is convex. To give you an intuitive idea about convexity, the figure titled $x^2 + y^2 \leq 1$ is *convex* 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.



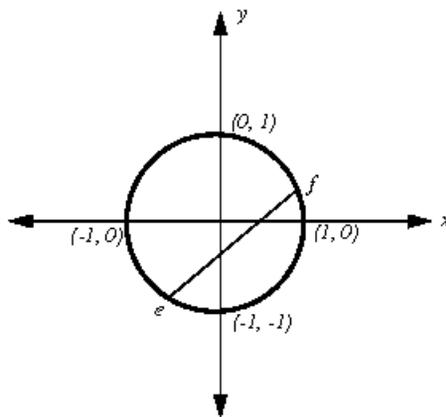
$x^2 + y^2 \leq 1$ is convex

The inequality $x^2 + y^2 \geq 1$ is not convex; it is concave. The figure titled $x^2 + y^2 \geq 1$ is *not convex* 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.



$x^2 + y^2 \geq 1$ 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 the figure titled $x^2 + y^2 = 1$ is not convex, 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.



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

Semi-definiteness

Identifying a quadratically constrained program (QCP) explained that the quadratic matrix in each constraint must be positive semi-definite (PSD), thus providing convexity. A matrix Q_i is PSD if $x^T Q_i x \geq 0$ for every vector x , whether or not x is feasible. Other issues pertaining to positive semi-definiteness are discussed in the context of a quadratic objective function in *Identifying convex QPs*.

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)*. 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:

$$-c_0x_0^2 + \sum c_i x_i^2 \leq 0$$

CPLEX automatically transforms quadratic constraints of these types:

```
x'Qx <= y2 where y >= 0 and Q is PSD
```

```
x'Qx <= yz where y >= 0 , z >= 0, and Q is PSD
```

into second order cone constraints.

Detecting the problem type of a QCP or SOCP

Documents the criteria that ILOG CPLEX components use to detect the problem type of a quadratically constrained program.

In this section

Overview

Offers general considerations about QCP problem type in each API of ILOG CPLEX.

Concert Technology and QCP problem type

Documents the role of the quadratic constraints in Concert Technology applications.

Callable Library and QCP problem type

Documents the role of quadratic constraints in the C API.

Interactive Optimizer and QCP problem type

Documents the role of quadratic constraints in the Interactive Optimizer.

File formats and QCP problem type

Describes the file formats that accommodate quadratic constraints.

Overview

The various components of ILOG CPLEX (the object-oriented APIs of Concert Technology, the C API of the Callable Library, and the Interactive Optimizer) give more or less weight to the idea of problem type. The following topics discuss the issues of problem type for each of those components with respect to QCPs and SOCPs.

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, just as other constraints are. When extracting a model with a quadratic constraint, ILOG CPLEX automatically detects it as a QCP and makes 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 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*.

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 state your quadratic constraints in the `subject to` section of the file. For more detail about representing QCP models in MPS file format, see the *ILOG CPLEX File Format Reference Manual*, especially the topic *Quadratically constrained programs (QCP) in MPS files*. 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 specify 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 `CPXchgproctype` 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 `CPXchgproctype` 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` specifies 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` specifies 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` specifies 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`* and about Java applications in *The active model*. To add constraints to a model in the .NET framework, see *ILOG Concert Technology for .NET users*.

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* and about Java applications in *Modifying the model*.

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*. 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 also change the problem type, as explained in *Changing problem type*.

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 Technology** 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.

However, in a mixed integer quadratically constrained programming (MIQCP) problem, you can specify whether ILOG CPLEX solves a QCP relaxation or LP relaxation of the subproblems. The *MIQCP strategy switch* parameter (`MIQCPStrat`, `CPX_PARAM_MIQCPSTRAT`) lets you specify which type of relaxation to solve.

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`

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.

In this section

Solving mixed integer programming problems (MIP)

Documents the solution of mixed integer programs (MIPs) with the ILOG CPLEX Mixed Integer Optimizer; that is, solving models in which one or more variables must take integer solution values.

Solution pool: generating and keeping multiple solutions

Introduces the solution pool for storing multiple solutions to a mixed integer programming problem (MIP) and explains techniques for generating and managing those solutions.

Using special ordered sets (SOS)

Describes special ordered sets (SOSs) in a model as a way to specify integrality conditions.

Using semi-continuous variables: a rates example

Demonstrates semi-continuous variables in Concert Technology in an example of managing production in a power plant.

Using piecewise linear functions in optimization: a transport example

Demonstrates the use of piecewise linear functions to solve a transportation problem.

Logical constraints in optimization

Describes logical constraints in ILOG CPLEX with Concert Technology.

Indicator constraints in optimization

Introduces indicator constraints and emphasizes their advantages over Big M formulations..

Using logical constraints: Food Manufacture 2

Demonstrates logical constraints in a sample application.

Early tardy scheduling

Solves a scheduling problem by applying logical constraints, piecewise linear functions, and aggressive MIP emphasis.

Using column generation: a cutting stock example

Uses an example of cutting stock to demonstrate the technique of column generation in Concert Technology.

Solving mixed integer programming problems (MIP)

Documents the solution of mixed integer programs (MIPs) with the ILOG CPLEX Mixed Integer Optimizer; that is, solving models in which one or more variables must take integer solution values.

In this section

Stating a MIP problem

Defines the kind of problems that the mixed integer optimizer solves.

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 types, displaying MIPs in the Interactive Optimizer, detecting the problem type, and switching to the fixed form of your problem.

Using the mixed integer optimizer

Describes features of the MIP optimizer.

Tuning performance features of the mixed integer optimizer

Describes features for tuning performance of the MIP optimizer.

Cuts

Describes types of cuts available in the MIP optimizer as performance features.

Heuristics

Introduces heuristics in performance features.

Preprocessing: presolver and aggregator

Describes preprocessing in the MIP optimizer.

Starting from a solution: MIP starts

Documents advanced starts; also known as warm starts or MIP starts.

Issuing priority orders

Describes priority order to control MIP optimization.

Using the MIP solution

Describes access to the solution of a MIP optimization.

Progress reports: interpreting the node log

Describes the progress reports issued in the node log during MIP optimization.

Troubleshooting MIP performance problems

Describes symptoms of performance problems in MIP optimization and recommends remedies.

Examples: optimizing a simple MIP problem

Introduces samples that demonstrate how to optimize a MIP with the ILOG CPLEX Component Libraries.

Example: reading a MIP problem from a file

Introduces samples that show to solve a MIP with the Component Libraries when the problem data is stored in a file.

$$0 \leq x_2 \leq +\infty$$

$$0 \leq x_3 \leq +\infty$$

$$2 \leq x_4 \leq 3$$

x_4 integer

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 types, displaying MIPs in the Interactive Optimizer, detecting the problem type, and switching to the fixed form of your problem.

In this section

Entering MIP problems

Describes special considerations for entering a MIP or reading a MIP from a file,

Displaying MIP problems

Describes options for displaying a MIP model.

Changing problem type in MIPs

Describes means of changing the type of a MIP model; also specifies how ILOG CPLEX determines the type of a MIP model.

Changing variable type

Describes special considerations about changing the type of a variable in a MIP model.

Entering MIP problems

You enter MIPs into ILOG CPLEX as explained in each of the topics about the APIs of ILOG CPLEX, with this additional consideration: you need to specify which variables are binary, general integer, semi-continuous, and semi-integer, and which are contained in special ordered sets (SOS).

Concert Technology users specify this information by passing the value of a type to the appropriate constructor when creating the variable, as summarized in *Specifying type of variable for a MIP in Concert Technology*.

Specifying type of variable for a MIP in Concert Technology

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 specify this information through the routine `CPXcopyctype`.

In the **Interactive Optimizer**, to specify binary integers in the context of the `enter` command, type `binaries` on a separate line, followed by the designated binary variables. To specify general integers, type `generals` on a separate line, followed by the designated general variables. To specify semi-continuous variables, type `semi-continuous` on a separate line, followed by the designated variables. Semi-integer variables are specified as both general integer and semi-continuous. The order of these three sections (`generals`, `semi-continuous`, `semi-integer` as both) does not matter. To enter the general integer variable of the *Stating a MIP problem*, you type this:

```
generals
x4
```

You may also read MIP data in from a formatted file, just as you do for linear programming problems. *Understanding file formats* in this manual lists the file formats briefly, and the *ILOG CPLEX File Formats 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

Interactive Optimizer display options for MIP problems summarizes display options in the **Interactive Optimizer** that are specific to MIP problems.

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 SOSs, if present.

In **Concert Technology**, use one of the accessors supplied with the appropriate object class, such as `IloSOS2::getVariables`.

From the **Callable Library**, use the routines `CPXgetctype` and `CPXgetsos` to access this information.

Changing problem type in MIPs

Concert Technology applications treat all models as capable of containing integer variables, and thus these variable declarations may be added or deleted at will. When extracting a model with integer variables, ILOG CPLEX in **Concert Technology** will automatically detect the model 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 declare the problem type with the Callable Library and the Interactive Optimizer are discussed in this topic.

When you enter a problem, ILOG CPLEX detects 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 discovered according to *Definitions of problem types*.

Definitions of problem types

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 modification is accomplished by first changing the problem type to milp, 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`

specifying 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, the Interactive Optimizer creates an empty quadratic matrix, ready for populating via the `change qpterm` command.

◆ `lp`, `qcp`, or `qp`

specifying 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, the Interactive Optimizer creates an empty quadratic matrix, 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 must 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

In the Interactive Optimizer, the command `change type` adds (or removes) the restriction on a variable that it must be an integer. 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 & 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 round-off.

Using the mixed integer optimizer

Describes features of the MIP optimizer.

In this section

Invoking the optimizer for a MIP model.

Describes invocation of the optimizer for a MIP model.

Emphasizing feasibility and optimality

Describes the context of the MIP emphasis parameter.

Terminating MIP optimization

Describes termination conditions of the MIP optimizer.

Invoking the optimizer for a MIP model.

The ILOG CPLEX Mixed Integer Optimizer solves MIP models using a very general and robust algorithm based on branch & cut. 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 method `solve`.
- ◆ In the **Callable Library**, use the routine `CPXmipopt`.

Emphasizing feasibility and optimality

The following topic, *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` or `CPX_PARAM_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 (that is, 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 may want a greater emphasis on feasibility and less emphasis on analysis and proof of optimality. For example, a restrictive time limit (set by the user with the `TimeLimit` 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 `CPX_MIPEMPHASIS_FEASIBILITY` in the Callable Library) 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`). This setting

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 with 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 with 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 search 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 by 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 parts of the search space 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 (`ObjDif`) 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 to allow early termination; for example, a relative MIP gap of 0.05 (corresponding to 5%). 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. *Parameters to limit MIP optimization* summarizes those parameters and their purpose.

Parameters to limit MIP optimization

To set a limit on	Use this parameter	Concert Technology	Callable Library	Interactive Optimizer
elapsed time	<i>optimizer time limit</i>	TiLim	CPX_PARAM_TILIM	timelimit
number of nodes	<i>MIP node limit</i>	NodeLim	CPX_PARAM_NODELIM	mip limits nodes
size of tree	<i>tree memory limit</i>	TreLim	CPX_PARAM_TRELIM	mip limits treememory
number of integer solutions	<i>MIP integer solution limit</i>	IntSolLim	CPX_PARAM_INTSOLLIM	mip limits solutions
relative MIP gap tolerance	<i>relative MIP gap tolerance</i>	EpGap	CPX_PARAM_EPGAP	mip tolerances mipgap

To set a limit on	Use this parameter	Concert Technology	Callable Library	Interactive Optimizer
absolute MIP gap tolerance	<i>absolute MIP gap tolerance</i>	EpAGap	CFX_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*.

Tuning performance features of the mixed integer optimizer

Describes features for tuning performance of the MIP optimizer.

In this section

Branch & cut or dynamic search?

Describes the search facilities of the MIP optimizer.

Introducing performance features of the MIP optimizer

Describes performance features generally.

Applying cutoff values

Describes conditions in which ILOG CPLEX applies a cutoff value.

Applying tolerance parameters

Describes conditions in which ILOG CPLEX applies tolerances.

Applying heuristics

Describes conditions in which ILOG CPLEX applies heuristics.

When an integer solution is found: the incumbent

Describes ILOG CPLEX handling of an incumbent value.

Controlling strategies: diving and backtracking

Describes parameters to control search strategies

Selecting nodes

Describes options of the parameter to control selection of strategies applied at nodes and their effect on the search.

Selecting variables

Describes options of the parameter to control selection of the next variable.

Changing branching direction

Describes options of the parameter to change branching direction and its effect on search.

Solving subproblems

Describes the parameters to control solution of subproblems.

Using node files

Describes a storage feature for nodes of the search.

Probing

Describes probing, a technique that fixes binary variables and analyzes the implications.

Branch & cut or dynamic search?

In addition to a robust branch & cut algorithm, ILOG CPLEX also offers a *dynamic search algorithm*. The dynamic search algorithm consists of the same building blocks as branch & cut: LP relaxation, branching, cuts, and heuristics. The following sections of the manual describe performance and tuning in relation to branch & cut. The parameters mentioned in this context have a similar effect in the dynamic search algorithm as in conventional branch & cut. In fact, the generic description in relation to branch & cut suffices conceptually for both branch & cut and dynamic search, even though their implementations differ.

ILOG CPLEX offers the MIP search parameter (*MIP dynamic search switch*: `MIPSearch`, `CPX_PARAM_MIPSEARCH`) for you to control whether it pursues dynamic search or conventional branch & cut in solving your problem. At its default setting, this parameter specifies that ILOG CPLEX should choose which algorithm to apply on the basis of characteristics it finds in your model. Other settings allow you to specify which search to pursue.

Because many parameter settings directly affect the branch & cut and dynamic search algorithms, here is a general description of how branch & cut 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* 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.

Introducing performance features of the MIP 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.

To help you decide whether default settings of parameters are best for your model, or whether other parameter settings may improve performance, the tuning tool is available. *Tuning tool* explains more about this utility and directs you to examples of its use.

This section discusses the ILOG CPLEX features and parameters that are the most likely to offer help with difficult models.

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 `ObjDiff` (absolute objective function difference) or `RelObjDiff` (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 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 inexpensively to 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 *Parameters for controlling branch & cut strategy*.

Parameters for controlling branch & cut strategy

Interactive Optimizer Command	Concert Technology IloCPLEX Method	Callable Library Routine	Parameter Reference
set mip strategy backtrack	setParam (BtTol , n)	CPXsetdblparam (env, CPX_PARAM_BTOL , n)	<i>backtracking tolerance</i>
set mip strategy nodeselect	setParam (NodeSel , i)	CPXsetintparam (env, CPX_PARAM_NODESEL , i)	<i>MIP node selection strategy</i>
set mip strategy variableselect	setParam (VarSel , i)	CPXsetintparam (env, CPX_PARAM_VARSEL , i)	<i>MIP variable selection strategy</i>
set mip strategy bbinterval	setParam (BBInterval , i)	CPXsetintparam (env, CPX_PARAM_BBINTERVAL , i)	<i>MIP strategy best bound interval</i>
set mip strategy branch	setParam (BrDir , i)	CPXsetintparam (env, CPX_PARAM_BRDIR , i)	<i>MIP branching direction</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 tolerance parameter, `BtTol` or `CPX_PARAM_BTOL`, influences this decision, in terms of the relative degradation of the objective function caused by the branches taken so far in this dive. Setting that parameter 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 makes 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 sets this choice. (For documentation of the node selection parameter, see *MIP node selection strategy* in the *ILOG CPLEX Parameters Reference Manual*.)

Node selection parameter settings for node search type

Value of the parameter	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 parameter known as *MIP strategy best bound interval* (BBinterval or CPX_PARAM_BBINTERVAL) sets 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 influences which variable is chosen for branching at that node. (For documentation of that parameter, see *MIP variable selection strategy* in the *ILOG CPLEX Parameters Reference Manual*.)

VarSel parameter settings for choice of branching variable

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 decides 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 branching direction parameter influences the direction, up or down, of the branch on that variable to be explored first. (For documentation of that parameter, see *MIP branching direction* in the *ILOG CPLEX Parameters Reference Manual*.)

BrDir parameter settings for choice of branching direction

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

Priority orders complement the behavior of these parameters. They are introduced in *Issuing priority orders*. 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 *MIP starting algorithm* parameter (`RootAlg`, `CPX_PARAM_STARTALG`) enables you to specify which algorithm for ILOG CPLEX to apply to the initial relaxation.

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

For more detail about situations in which you may find these parameters helpful, see *Unsatisfactory optimization of subproblems*.

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. Storage of 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*.

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 probing parameter (0 (zero)), ILOG CPLEX will automatically decide an appropriate level of probing. Setting the probing parameter to 1, 2, or 3, results in increasing levels of probing performed beyond the default level. A setting of -1 results in no probing being performed.

To activate an increasing level of probing use the *MIP probing level* parameter:

- ◆ 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

Describes types of cuts available in the MIP optimizer as performance features.

In this section

What are cuts?

Defines cuts.

Clique cuts

Defines a clique cut.

Cover cuts

Defines a cover cut.

Disjunctive cuts

Defines disjunctive cuts.

Flow cover cuts

Defines flow cover cuts.

Flow path cuts

Defines flow path cuts.

Gomory fractional cuts

Defines Gomory fractional cuts.

Generalized upper bound (GUB) cover cuts

Defines GUB cover cuts.

Implied bound cuts

Defines implied bound cuts.

Mixed integer rounding (MIR) cuts

Defines MIR cuts.

Zero-half cuts

Defines zero-half cuts and offers an example.

Adding cuts and re-optimizing

Describes conditions under which ILOG CPLEX adds cuts.

Counting cuts

Describes methods and routines to calculate how many cuts have been added.

Parameters affecting cuts

Summarizes parameters controlling cuts.

What are 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.

Here are descriptions of the cuts implemented in CPLEX:

- ◆ *Clique cuts*
- ◆ *Cover cuts*
- ◆ *Disjunctive cuts*
- ◆ *Flow cover cuts*
- ◆ *Flow path cuts*
- ◆ *Gomory fractional cuts*
- ◆ *Generalized upper bound (GUB) cover cuts*
- ◆ *Implied bound cuts*
- ◆ *Mixed integer rounding (MIR) cuts*
- ◆ *Zero-half cuts*

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.

Zero-half cuts

Zero-half cuts are based on the observation that when the lefthand side of an inequality consists of integral variables and integral coefficients, then the righthand side can be rounded down to produce a zero-half cut. To understand how zero-half cuts are generated, consider these two constraints over five integer variables with integer coefficients:

$$\begin{array}{rcl} x_1 + 2x_2 + x_3 + 3x_4 & \leq & 8 \\ x_1 + 3x_3 + x_4 + 2x_5 & \leq & 5 \end{array}$$

Now consider the sum of those two constraints:

$$2x_1 + 2x_2 + 4x_3 + 4x_4 + 2x_5 \leq 13$$

Divide that constraint by 2:

$$x_1 + x_2 + 2x_3 + 2x_4 + x_5 \leq 6.5$$

Round down the righthand side to get the zero-half cut:

$$x_1 + x_2 + 2x_3 + 2x_4 + x_5 \leq 6$$

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

Cuts may be added to a problem during MIP optimization. After MIP optimization terminates, to know the number of cuts that were added during optimization, implement one of the following techniques in your application:

- ◆ For **Concert Technology**, use the method:
 - `IloCplex::getNcuts` in the C++ API;
 - `IloCplex.getNcuts` in the Java API;
 - `Cplex.GetNcuts` in the .NET API.
- ◆ For **Callable Library**, use the routine `CPXgetnumcuts`.

Tip: After MIP optimization terminates, those techniques count the number of cuts that were added **during** the optimization. In order to count cuts **while** optimization is on going, consider methods from the **query callbacks**, such as `IloCplex::MIPCallback::getNzeroHalfCut`, which counts the number of zero-half cuts, for example. For more detail, see *Query or diagnostic callbacks*, which tells where to find examples of their use and where to find documentation of such methods in the reference manuals of the APIs.

Parameters affecting cuts

Parameters control the way each class of cuts is used. Those parameters are listed in the table *Parameters for controlling cuts*.

Parameters for controlling cuts

Cut Type	Interactive Command	ConcertTechnology	Callable Library	Parameter Reference
Clique	set mip cuts cliques	Cliques	CPX_PARAM_CLIQUES	<i>MIP cliques switch</i>
Cover	set mip cuts covers	Covers	CPX_PARAM_COVERS	<i>MIP covers switch</i>
Disjunctive	set mip cuts disjunctive	DisjCuts	CPX_PARAM_DISJCUTS	<i>MIP disjunctive cuts switch</i>
Flow Cover	set mip cuts flowcovers	FlowCovers	CPX_PARAM_FLOWCOVERS	<i>MIP flow cover cuts switch</i>
Flow Path	set mip cuts pathcut	FlowPaths	CPX_PARAM_FLOWPATHS	<i>MIP flow path cut switch</i>
Gomory	set mip cuts gomory	FracCuts	CPX_PARAM_FRACCUTS	<i>MIP Gomory fractional cuts switch</i>
GUB Cover	set mip cuts gubcovers	GUBCovers	CPX_PARAM_GUBCOVERS	<i>MIP GUB cuts switch</i>
Implied Bound	set mip cuts implied	ImplBd	CPX_PARAM_IMPLBD	<i>MIP implied bound cuts switch</i>
Mixed Integer Rounding (MIR)	set mip cuts mircut	MIRCuts	CPX_PARAM_MIRCUTS	<i>MIP MIR (mixed integer rounding) cut switch</i>
Zero-Half	set mip cuts zerohalfcut	ZeroHalfCuts	CPX_PARAM_ZEROHALFCUTS	<i>MIP zero-half cuts switch</i>

The default value of each of those parameters is 0 (zero). By default, ILOG CPLEX automatically decides 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 simultaneously.

The *row multiplier factor for cuts* (`CutsFactor`, `CPX_PARAM_CUTSFACTOR`) parameter controls the number of cuts ILOG CPLEX adds to the model. The problem can grow to `CutsFactor` times the original number of rows in the model (or in the presolved model, if the presolver is active). Thus, a `CutsFactor` of 1.0 would mean that no cuts will be generated, which may be a more convenient way of turning off all cuts than setting them individually. The default `CutsFactor` value of 4.0 works well in most cases, as it allows a generous number of cuts while in rare instances it also serves to limit unchecked growth in the problem size.

The *constraint aggregation limit for cut generation* (`AggCutLim`, `CPX_PARAM_AGGCUTLIM`) parameter controls the number of constraints allowed to be aggregated for generating MIR and flow cover cuts.

The *pass limit for generating Gomory fractional cuts* (`FracPass`, `CPX_PARAM_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 *candidate limit for generating Gomory fractional cuts* (`FracCand`, `CPX_PARAM_FRACCAND`) parameter controls the number of variable candidates to be considered for generating Gomory fractional cuts.

Heuristics

Introduces heuristics in performance features.

In this section

What are heuristics?

Defines a heuristic and describes conditions under which ILOG CPLEX applies heuristics in MIP optimization.

Node heuristic

Describes the heuristic applied at nodes by the MIP optimizer.

Relaxation induced neighborhood search (RINS) heuristic

Describes RINS as a heuristic of the MIP optimizer.

Solution polishing

Describes solution polishing as a heuristic of the MIP optimizer.

Feasibility pump

Describes the feasibility pump as a heuristic of the MIP optimizer.

What are 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.

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.

ILOG CPLEX provides families of heuristics to find integer solutions at nodes (including the root node) during the branch & cut procedure. These families of heuristics are documented in the following topics.

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 (zero), ILOG CPLEX dynamically sets 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.

For more about this heuristic, see the article published by Emilie Danna, Edward Rothberg, Claude Le Pape in 2005, titled *Exploring relaxation induced neighborhoods to improve MIP solutions* in the journal **Mathematical Programming** in volume 102, issue 1, pages 71–90.

Solution polishing

Solution polishing can yield better solutions in situations where good solutions are otherwise hard to find. More time-intensive than other heuristics, solution polishing is actually a variety of branch & cut that works after an initial solution is available. In fact, it requires a solution to be available for polishing, either a solution produced by branch & cut, or a MIP start supplied by a user.

Because of the high cost entailed by solution polishing, it is not called throughout branch & cut like other heuristics. Instead, solution polishing works in a second phase after a first phase of conventional branch & cut. As an additional step after branch & cut, solution polishing can improve the best known solution.

As a kind of branch & cut algorithm itself, solution polishing focuses solely on finding better solutions. Consequently, it may not prove optimality, even if the optimal solution has indeed been found.

Stopping criteria for solution polishing

Solution polishing obeys the same stopping criteria as branch & cut.

- ◆ The absolute gap tolerance is a stopping criterion for polishing. For more general information about it, see *absolute MIP gap tolerance* in the *ILOG CPLEX Parameters Reference Manual*.
 - EpAGap in Concert Technology
 - CPX_PARAM_EPAGAP in the Callable Library
 - `mip tolerances absmipgap` in the Interactive Optimizer
- ◆ The relative gap tolerance is a stopping criterion for polishing. For more general information about it, see *relative MIP gap tolerance* in the *ILOG CPLEX Parameters Reference Manual*.
 - EpGap in Concert Technology
 - CPX_PARAM_EPGAP in the Callable Library
 - `mip tolerances mipgap` in the Interactive Optimizer
- ◆ The optimizer time limit is a stopping criterion for polishing. For more general information about it, see *optimizer time limit* in the *ILOG CPLEX Parameters Reference Manual*.
 - TiLim in Concert Technology
 - CPX_PARAM_TILIM in the Callable Library
 - `timelimit` in the Interactive Optimizer
- ◆ The node limit is a stopping criterion for polishing. For more general information about it, see *MIP node limit* in the *ILOG CPLEX Parameters Reference Manual*.

- NodeLim in Concert Technology
- CPX_PARAM_NODELIM in the Callable Library
- mip limits nodes in the Interactive Optimizer
- ◆ The integer solution limit is a stopping criterion for polishing. For more general information about it, see *MIP integer solution limit* in the *ILOG CPLEX Parameters Reference Manual*.
 - IntSolLim in Concert Technology
 - CPX_PARAM_INTSOLLIM in the Callable Library
 - mip limits solutions in the Interactive Optimizer

Those criteria apply to the overall optimization, that is, branch & cut and solution polishing. For example, if you set the *optimizer time limit* (TiLim, CPX_PARAM_TILIM) to 100 seconds, then CPLEX spends at most 100 seconds in total for branch & cut and solution polishing.

Starting conditions for solution polishing

You control when CPLEX switches from branch & cut to solution polishing with these parameters:

- ◆ The *absolute MIP gap before starting to polish a feasible solution*
 - PolishAfterEpAGap in Concert Technology
 - CPX_PARAM_POLISHAFTEREPAGAP in the Callable Library
 - mip polishafter absmipgap in the Interactive Optimizer
- ◆ The *relative MIP gap before starting to polish a feasible solution*
 - PolishAfterEpGap in Concert Technology
 - CPX_PARAM_POLISHAFTEREPGAP in the Callable Library
 - mip polishafter mipgap in the Interactive Optimizer
- ◆ The number of *MIP integer solutions to find before starting to polish a feasible solution*
 - PolishAfterIntSol in Concert Technology
 - CPX_PARAM_POLISHAFTERINTSOL in the Callable Library
 - mip polishafter solutions in the Interactive Optimizer
- ◆ The number of *nodes to process before starting to polish a feasible solution*
 - PolishAfterNode in Concert Technology
 - CPX_PARAM_POLISHAFTERNODE in the Callable Library

- `mip polishafter` nodes in the Interactive Optimizer
- ◆ The amount (in seconds) of *time before starting to polish a feasible solution*
 - `PolishAfterTime` in Concert Technology
 - `CPX_PARAM_POLISHAFTERTIME` in the Callable Library
 - `mip polishafter time` in the Interactive Optimizer

With each of those parameters, a user tells CPLEX when to switch from branch & cut to solution polishing. CPLEX is able to switch after it has found a feasible solution **and** put into place the MIP structures needed for solution polishing.

When these two conditions are met (feasible solution and structures in place), CPLEX stops branch & cut and switches to solution polishing whenever the **first** of these starting conditions is met:

- ◆ when CPLEX achieves a specified absolute MIP gap;
- ◆ when CPLEX achieves a specified relative MIP gap;
- ◆ when CPLEX finds a specified number of integer feasible solutions;
- ◆ when CPLEX processes a specified number of nodes;
- ◆ when CPLEX has spent a specified amount of time in optimization.

Other parameters governing solution polishing

Like the RINS heuristic, solution polishing explores neighborhoods of previously found solutions by solving subMIPs. Consequently, the subMIP node limit also controls aspects of the work that solution polishing performs. See the *limit on nodes explored when a subMIP is being solved* in the *ILOG CPLEX Parameters Reference Manual*.

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

Solution polishing operates in a second phase, **after** a first phase of the usual branch & cut, but it operates within the same branch & cut framework. Consequently, the same parameters that govern branch & cut also govern solution polishing.

For example, if multiple threads are available in your application, solution polishing can exploit them to work in parallel. See *global default thread count* in the *ILOG CPLEX Parameters Reference Manual*.

- ◆ `Threads` in Concert Technology
- ◆ `CPX_PARAM_THREADS` in the Callable Library
- ◆ `threads` in the Interactive Optimizer

Similarly, your choice of opportunistic or deterministic parallel mode for MIP optimization also governs solution polishing. For more detail about choosing opportunistic or deterministic parallel mode, see *parallel mode switch* in the *ILOG CPLEX Parameters Reference Manual*.

- ◆ `ParallelMode` in Concert Technology
- ◆ `CPX_PARAM_PARALLELMODE` in the Callable Library
- ◆ `parallel` in the Interactive Optimizer

Callbacks and solution polishing

Callbacks are valid and work during solution polishing. However, nodes are processed much more slowly during solution polishing because of the more expensive work carried out at each node. Consequently, callbacks may be called less often during solution polishing.

Finding a first solution to improve by polishing

A typical use of solution polishing is first to find a preliminary solution with branch & cut and then to improve it with solution polishing.

To create conditions where you can find a first solution and then improve it by means of polishing, follow these steps:

1. Set to 1 (one) the number of *MIP integer solutions to find before starting to polish a feasible solution*.
 - ◆ `PolishAfterIntSol` in Concert Technology
 - ◆ `CPX_PARAM_POLISHAFTERINTSOL` in the Callable Library
 - ◆ `mip polishafter solutions` in the Interactive Optimizer
2. Set the *optimizer time limit* to a positive value, such as 200 seconds, to specify the total time for CPLEX to spend in branch & cut and polishing.
3. Call the optimizer.

Improving a MIP start with polishing

Similarly, polishing can improve a MIP start.

To create conditions to improve a MIP start with polishing, follow these steps:

1. Set to 0 (zero) the *time before starting to polish a feasible solution*.
2. Set the *optimizer time limit* to a positive number of seconds.
 - ◆ `TiLim` in Concert Technology
 - ◆ `CPX_PARAM_TILIM` in the Callable Library
 - ◆ `timelimit` in the Interactive Optimizer
3. Verify that the *advanced start switch* remains at its default value of 1 (one) so that polishing will proceed from the MIP start.

- ◆ AdvInd in Concert Technology
- ◆ CPX_PARAM_ADVIND in the Callable Library
- ◆ advance in the Interactive Optimizer

4. Specify the MIP start, for example, by reading it from a file or adding it to the model, as explained in *Establishing starting values in a MIP start*.
5. Call the optimizer.

With those settings, CPLEX switches to polishing as soon as all MIP structures needed for polishing are in place. It does **not** completely solve the root node with those settings. In particular, it does **not** generate cuts under these conditions.

If CPLEX is unable to process the MIP start into a solution, then solution polishing does not begin until after branch & cut finds a solution.

In contrast, if you want to **solve the root node**, and if the MIP start has been processed into a solution, you must change that step 1 before applying the other steps:

1. Set the starting condition for polishing by means of this parameter, the number of *nodes to process before starting to polish a feasible solution*, set to 1 (one).
 - ◆ PolishAfterNode in Concert Technology
 - ◆ CPX_PARAM_POLISHAFTERNODE in the Callable Library
 - ◆ mip_polishafter nodes in the Interactive Optimizer
2. Set the *optimizer time limit* to a positive number of seconds.
3. Verify that the *advanced start switch* remains at its default value of 1 (one).
4. Specify the MIP start, for example, by reading it from a file or adding it to the model, as explained in *Establishing starting values in a MIP start*.
5. Call the optimizer.

If your model and application are such that processing the root node takes too much time, try the recommendations in these other topics:

- ◆ *Large number of unhelpful cuts*
- ◆ *Too much time at node 0*

In the rare event that solution polishing is unable to improve a MIP start that you provide, polishing may be more successful if you disable some combination of dual reductions, nonlinear reductions, or symmetry reductions during preprocessing.

For details about the parameters to disable those features, see the *ILOG CPLEX Parameter Reference Manual*, especially these topics:

- ◆ the *presolve switch*: `PreInd`, `CPX_PARAM_PREIND`
- ◆ the *linear reduction switch*: `PreLinear`, `CPX_PARAM_PRELINEAR`
- ◆ the *symmetry breaking* parameter: `Symmetry`, `CPX_PARAM_SYMMETRY`

Controlling solution polishing with time as a criterion

As an example to illustrate how to manage time spent polishing a feasible solution, suppose the user wants to solve a problem by spending 100 seconds in branch & cut and 200 seconds in polishing.

1. Set the *optimizer time limit* to 300.0 seconds. This parameter controls the total time spent in branch & cut and solution polishing.
 - ◆ `TiLim` in Concert Technology
 - ◆ `CPX_PARAM_TILIM` in the Callable Library
 - ◆ `timelimit` in the Interactive Optimizer
2. Set to 100 seconds the amount of *time before starting to polish a feasible solution* as the starting condition for polishing. This parameter controls the amount of time CPLEX spends in branch & cut before CPLEX switches to solution polishing.
 - ◆ `PolishAfterTime` in Concert Technology
 - ◆ `CPX_PARAM_POLISHAFTERTIME` in the Callable Library
 - ◆ `mip polishafter time` in the Interactive Optimizer
3. Call the optimizer.

Under those conditions, if CPLEX finds a feasible solution within the first 100 seconds of branch & cut, then it switches to solution polishing after exactly 100 seconds. However, if CPLEX does not find a feasible solution within the first 100 seconds, then it continues branch & cut until it finds a first feasible solution and switches to solution polishing afterward.

Those settings guarantee that CPLEX spends at most 300 seconds on the model and that CPLEX applies polishing only if it finds a feasible solution within that time.

Controlling solution polishing with a gap as a criterion

CPLEX also allows you to control when polishing starts and when polishing ends with a gap as a criterion. The gap may be relative or absolute. For example, suppose you want to apply branch & cut until achieving a 10% relative gap and then you want to switch to solution polishing until achieving a 2% relative gap.

To apply branch & cut until achieving a 10% gap and then to switch to solution polishing until achieving a 2% gap, follow these steps:

1. Set the *relative MIP gap tolerance* to 2%. This parameter determines when overall optimization stops.

- ◆ EpGap in Concert Technology
 - ◆ CPX_PARAM_EPGAP in the Callable Library
 - ◆ mip tolerances mipgap in the Interactive Optimizer
2. Set to 10% the *relative MIP gap before starting to polish a feasible solution*. This parameter sets the starting condition for polishing. It controls when CPLEX switches from branch & cut to solution polishing.
 3. Set the *optimizer time limit* to a positive value (for example, 200 seconds) to specify the total time spent in branch & cut plus polishing. For difficult problems, this step is a precaution to guarantee that CPLEX terminates even if the targeted gap cannot be achieved.
 - ◆ TiLim in Concert Technology
 - ◆ CPX_PARAM_TILIM in the Callable Library
 - ◆ timelimit 200.0 in the Interactive Optimizer
 4. Call the optimizer.

For more about solution polishing

For technical detail about how solution polishing works, see the article by Edward Rothberg, *An evolutionary algorithm for polishing mixed integer programming solutions*, published in **INFORMS Journal on Computing**, volume 19, issue 4, pages 534–541 (2007).

Feasibility pump

The **feasibility pump** is a heuristic that finds an initial feasible solution even in certain very hard mixed integer programming problems (MIPs). In ILOG CPLEX, you control this heuristic by means of a parameter, *feasibility pump switch*, that specifies how the feasibility pump looks for a feasible solution.

- ◆ FBHeur in Concert Technology
- ◆ CPX_PARAM_FPHEUR in the Callable Library
- ◆ mip strategy fpheur in the Interactive Optimizer

Various settings of this parameter turn off the feasibility pump entirely, allow ILOG CPLEX to determine whether to apply the feasibility pump, emphasize finding any feasible solution, or emphasize finding a feasible solution with a good objective value. For more detail about this parameter and its settings, see *feasibility pump switch* in the Parameters Reference Manual.

For a theoretical and practical explanation of the feasibility pump heuristic, see research reported by Fischetti, Glover, and Lodi (2003, 2005), by Bertacco, Fischetti, and Lodi (2005), and by Achterberg and Berthold (2005, 2007).

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. *Parameters for controlling MIP preprocessing* 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.

Parameters for controlling MIP preprocessing

Interactive Command	Concert Technology Parameter	Callable Library Parameter	Comment	Parameter Reference
<code>set preprocessing aggregator</code>	AggInd	CPX_PARAM_AGGIND	on by default	<i>preprocessing aggregator application limit</i>
<code>set preprocessing presolve</code>	PreInd	CPX_PARAM_PREIND	on by default	<i>presolve switch</i>
<code>set preprocessing boundstrength</code>	BndStrenInd	CPX_PARAM_BNDSTRENGTH	presolve must be on	<i>bound strengthening switch</i>
<code>set preprocessing coeffreduce</code>	CoeRedInd	CPX_PARAM_COEREDIND	presolve must be on	<i>coefficient reduction setting</i>
<code>set preprocessing relax</code>	RelaxPreInd	CPX_PARAM_RELAXPREIND	applies to relaxation	<i>relaxed LP presolve switch</i>
<code>set preprocessing reduce</code>	Reduce	CPX_PARAM_REDUCE	all on by default	<i>primal and dual reduction type</i>
<code>set preprocessing numpass</code>	PrePass	CPX_PARAM_PREPASS	automatic by default	<i>limit on the number of presolve passes made</i>
<code>set preprocessing repeat</code>	RepeatPresolve	CPX_PARAM_REPEATPRESOLVE	automatic by default	<i>MIP repeat presolve switch</i>

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* 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 & 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 *MIP repeat presolve switch* (`RepeatPresolve`, `CPX_PARAM_REPEATPRESOLVE`) parameter 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. *Values of RepeatPresolve parameter* summarizes the possible values of this parameter.

Values of RepeatPresolve parameter

Value	Effect
-1	Automatic (default)
0	Turn off repeat presolve
1	Repeat presolve without cuts
2	Repeat presolve with cuts

Value	Effect
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: MIP starts

When you are solving a mixed integer programming problem (MIP), you can supply hints to help CPLEX find an initial solution. These hints consist of pairs of variables and values, known as a **MIP start**, an **advanced start**, or a **warm start**. A MIP start might come from a different problem you have previously solved or from your knowledge of the problem, for example. You can also provide CPLEX with one or more MIP starts, that is, multiple MIP starts.

A MIP start may be a feasible solution of the model, but it need not be; it may even be infeasible or incomplete. If you are interested in debugging an infeasible MIP start, that is, if you want to discover why CPLEX regards a MIP start as infeasible, consider using the conflict refiner on that MIP start, as explained in *Refining a conflict in a MIP start*.

What are the characteristics of a MIP start?

A MIP start may include continuous and discrete variables of various types, such as integer variables, binary variables, semi-continuous variables or those in lazy constraints, linearized constraints, or special ordered sets. For more information about each of those types, see these topics in this manual:

- ◆ *Using semi-continuous variables: a rates example*
- ◆ *User-cut and lazy-constraint pools*
- ◆ *Indicator constraints in optimization or Logical constraints in optimization*
- ◆ *Using special ordered sets (SOS)*

You may specify values for any combination of discrete and continuous variables. CPLEX decides how to construct a starting point from these variable-value pairs depending on the specified values and on the specified effort level. For more detail about effort level, see *MIP starts and effort level*.

Managing a MIP start with the advanced start switch

After a MIP start has been established for your model, you control its use by the *advanced start switch* (`AdvInd` in Concert Technology; `CPX_PARAM_ADVIND` in the Callable Library). At the default setting of 1 (one), 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. If you set this parameter to 2, ILOG CPLEX retains the current incumbent (if there is one), re-applies presolve, and starts a new search from a new root. Setting 2 can be particularly useful for solving fixed MIP models, where a start vector but no corresponding basis is available. For more about a fixed MIP, see *Working with the fixed MIP problem*.

Using a MIP start

When you provide a MIP start as data, CPLEX processes it before starting branch & cut during an optimization. If one or more of the MIP starts define a solution, CPLEX installs the best of these solutions as the **incumbent** solution. Having an incumbent from the very beginning of branch & cut allows CPLEX to eliminate portions of the search space and thus may result in smaller branch & cut trees. Having an incumbent also allows CPLEX to use heuristics which require an incumbent, such as relaxation induced neighborhood search (RINS heuristic) or solution polishing.

You may invoke relaxation induced neighborhood search on a starting solution. See *Relaxation induced neighborhood search (RINS) heuristic* in this manual for more about that topic.

Alternatively, you may invoke solution polishing to improve a solution known to CPLEX. See *Solution polishing*, also in this manual, for more about that way of proceeding.

Establishing starting values in a MIP start

You can establish MIP starting values by using the method `addMIPStart` in a **Concert Technology** application, or by using `CPXaddmipstarts` in a **Callable Library** application.

For use in the **Interactive Optimizer**, or as an alternative approach in a Concert Technology or Callable Library application, you can establish MIP starting values in a formatted file and then read the file.

To read a MIP start from a formatted file, use one of these:

- ◆ the method `readMIPStart` in **Concert Technology**
- ◆ the routine `CPXreadcopymipstarts` in the **Callable Library**
- ◆ the command `read` in the **Interactive Optimizer**

You can establish MIP starting values from a file in either MST or SOL format. MST and SOL share the same underlying XML format. MST format is documented in *MST file format: MIP starts* in the reference manual *ILOG CPLEX File Formats*. SOL format is documented in *SOL file format: solution files* also in the reference manual *ILOG CPLEX File Formats*.

Creating a file for a MIP start

At the end of a MIP optimization, when a feasible (not necessarily optimal) solution is still in memory, you can create an MST file for later use as starting values to another MIP problem.

- ◆ from **Concert Technology** with the method `writeMIPStarts`
- ◆ from the **Callable Library** with the routine `CPXwritemipstarts`
- ◆ from the **Interactive Optimizer** with the `write` command

Tip: Make sure that the name of each variable is consistent between the original model and your target model when you use this approach.

When you tell ILOG CPLEX to write a MIP start to a formatted file, you can also specify a degree of detail to record there, such as only values of discrete variables, values of all variables, and so forth. The *write level for MST, SOL files* is a parameter (`WriteLevel`, `CPX_PARAM_WRITELEVEL`), documented in the *ILOG CPLEX Parameter Reference Manual*, to manage the level of detail.

When ILOG CPLEX reads from such a file, it processes **all** the MIP starts. They will be processed at the next MIP optimization. Immediately after an optimization, the first MIP start is the MIP start corresponding to the incumbent solution, so if you write a file with multiple MIP starts, the **first** MIP start will be that corresponding to the **incumbent**.

Because processing a large number of MIP starts may be costly, CPLEX allows you to associate an individual **effort level** with each MIP start. The effort level tells CPLEX how to expend its effort in processing that MIP start. For more detail about effort level, see *MIP starts and effort level*.

MIP starts and effort level

You may want CPLEX to process multiple MIP starts differently, expending more effort on some than on others. Moreover, you may want to limit the effort CPLEX applies to MIP starts when it transforms each MIP start into a feasible solution, especially if there are many of them. In that context, you can specify a level of effort that CPLEX should expend for each MIP start to transform it into a feasible solution.

You specify the level of effort as an argument to the method or routine that adds a MIP start to a model or that modifies a MIP start. When CPLEX writes MIP starts to a file, such as a file in MST format, CPLEX records the level of effort the user specified for each MIP start. If you have not specified an effort level, CPLEX assigns a default effort level.

Here are the levels of effort and their effect:

- ◆ Level 1: CPLEX checks **feasibility** of the corresponding MIP start. This level requires you to provide values for **all** variables in the problem, both discrete and continuous. If any values are missing, CPLEX does not process this MIP start.
- ◆ Level 2: CPLEX solves the **fixed LP** problem specified by the MIP start. This effort level requires you to provide values for **all discrete** variables. If values for any discrete variables are missing, CPLEX does not process the MIP start. CPLEX uses the values of continuous variables at this effort level only in the initial phase when it attempts to determine values for unspecified discrete variables.

- ◆ Level 3: CPLEX solves a subMIP. You must specify the value of at least one discrete variable at this effort level. CPLEX uses the values of continuous variables at this effort level only in the initial phase when it attempts to determine values for unspecified discrete variables.
- ◆ Level 4: CPLEX attempts to repair the MIP start if it is infeasible, according to the parameter that sets the *frequency to try to repair infeasible MIP start* (RepairTries, CPX_PARAM_REPAIRTRIES). You must specify the value of at least one discrete variable at this effort level, too.

You may specify a different level of effort for each MIP start, for example, differing levels of effort for the incumbent, for a MIP start corresponding to a solution in the solution pool, for a MIP start supplied by the user. By **default**, CPLEX expends effort at level 4 for the first MIP start and at level 1 (one) for other MIP starts. You may change that level of effort; you do so by means of an argument to the method or routine when you add a MIP start to a model or when you modify the MIP start.

MIP starts and repair tries

If the values specified in your MIP start do not lead directly to an integer-feasible solution, CPLEX applies a heuristic to try to repair the MIP start. The frequency at which CPLEX applies the heuristic (that is, the number of branch & cut nodes processed between calls of the heuristic) is controlled by a parameter, the *frequency to try to repair infeasible MIP start* (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.

MIP starts and the solution pool

If you are solving a sequence of problems, for example, by modifying and re-solving the model, CPLEX creates a MIP start corresponding to the incumbent and to each member of the solution pool. You may add other MIP starts which you have computed. CPLEX then automatically processes all of these MIP starts, unless you have turned off MIP starts by setting that parameter to 0 (zero). For documentation of the MIP start parameter (AdvInd, CPX_PARAM_ADVIND) see *advanced start switch* in the *ILOG CPLEX Parameters Reference Manual*.

MIP starts and the Interactive Optimizer

To write a particular MIP start to a file from the Interactive Optimizer, use the `write` command supplying the file name, the file extension for MST formatted files, and the identifier of the MIP start, like this:

```
write filename.mst id
```

The identifier of the MIP start may be its **name** or its **index** number. In the Interactive Optimizer, MIP starts are named by default like this: m1, m2, m3, and so forth (that is, m followed by a number). The index number of a MIP start ranges from 1 (one) through the number of existing MIP starts for the current problem.

To **write all** existing MIP starts from the current session of the Interactive Optimizer to a formatted file, use this command:

```
write filename.mst all
```

To **delete** a MIP start from the current session of the Interactive Optimizer, use this command, where *id* is the name or index of the MIP start to delete:

```
change delete mipstart id
```

To delete a range of MIP starts, supply one the conventional options for a range in the Interactive Optimizer, using hyphen or wild card star, like these examples:

```
change delete mipstart 5-7
```

```
change delete mipstart *
```

Issuing priority orders

In the search, 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.

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.

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 *Parameter settings for branching priority order*.

Parameter settings for branching priority order

Value	Branching priority order
0	no automatic priority order will be generated (default)
1	decreasing cost coefficients among the variables

Value	Branching priority order
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 this parameter.

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 ILOG CPLEX applies a priority order, if one has been read in.

Using the MIP solution

This section discusses the **optimal** solution or the **best feasible** solution, if no optimum has been proved. For information about managing the entire pool of feasible solutions, see *Solution pool: generating and keeping multiple solutions*.

Accessing a MIP solution as values in an array

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.

Displaying a MIP solution in the Interactive Optimizer

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 rename the log file temporarily. 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
```

Accessing MIP solution information

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*.

For any of the MIP problem types, the following additional solution information is available in the Interactive Optimizer through options of 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.

Working with the fixed MIP problem

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 that applies only to the continuous variables, in the usual way. This observation about the information available only for continuous variables in a MIP is the idea behind the so-called "**fixed MIP**" problem. The fixed MIP is a form of the MIP problem where all of the discrete variables are placed at values corresponding to the MIP solution; that is, the discrete variables are fixed in the sense of set at a given value. Thus a fixed MIP 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`

As explained in *Managing a MIP start with the advanced start switch*, setting 2 of the *advanced start switch* (`AdvInd` in Concert Technology; `CPX_PARAM_ADVIND` in the Callable Library) can be particularly useful for solving fixed MIP models, where a start vector but no corresponding basis is available.

Progress reports: interpreting the node log

As explained in other topics, 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. The table *Values of the MIP display parameter* summarizes its possible values and their effects.

Values of the MIP display parameter

Value	Effect
0	No display until optimal solution has been found
1	Display integer feasible solutions
2	Display integer feasible solutions plus an entry for every n-th node; default
3	Display integer feasible solutions, every n-th node entry, number of cuts added, and information about the processing of each successful MIP start
4	Display integer feasible solutions, every n-th node entry, number of cuts added, information about the processing of each successful MIP start, and information about the LP subproblem at root
5	Display integer feasible solutions, every n-th node entry, number of cuts added, information about the processing of each successful MIP start, and information about the LP subproblem at root and at nodes

The *MIP node log interval* parameter (`MIPInterval`, `CPX_PARAM_MIPINTERVAL`) controls how frequently node log lines are printed. Its default is 100; it can be set to any positive integer value, as documented in the *ILOG CPLEX Parameters Reference Manual*. ILOG CPLEX records a line in the node log for every node with an integer solution if the parameter controlling *MIP node log display information* (`MIPDisplay`, `CPX_PARAM_MIPDISPLAY`) is set to 1 (one) or higher. Likewise, if the `MIPDisplay` is set to 2 or higher, then for any node whose number is a multiple of the `MIPInterval` value, a line is recorded in the node log for every node with an integer solution.

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.  
Presolve time = 0.00 sec.
```

MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
Parallel mode: none, using 1 thread.
Root relaxation solution time = 0.00 sec

	Node	Nodes Left	Objective	IInf	Best Integer	Cuts/ Best Node	ItCnt	Gap
*	0+	0			0.0000	3261.8212	8	--
-								
*	0+	0			3148.0000	3261.8212	8	3.
62%								
	0	0	3254.5370	7	3148.0000	Cuts: 5	14	3.
38%								
	0	0	3246.0185	7	3148.0000	Cuts: 3	24	3.
11%								
*	0+	0			3158.0000	3246.0185	24	2.
79%								
	0	0	3245.3465	9	3158.0000	Cuts: 5	27	2.
77%								
	0	0	3243.4477	9	3158.0000	Cuts: 5	32	2.
71%								
	0	0	3242.9809	10	3158.0000	Covers: 3	36	2.
69%								
	0	0	3242.8397	11	3158.0000	Covers: 1	37	2.
69%								
	0	0	3242.7428	11	3158.0000	Cuts: 3	39	2.
68%								
	0	2	3242.7428	11	3158.0000	3242.7428	39	2.
68%								
	10	11	3199.1875	2	3158.0000	3215.1261	73	1.
81%								
*	20+	11			3168.0000	3215.1261	89	1.
49%								
	20	13	3179.0028	5	3168.0000	3215.1261	89	1.
49%								
	30	15	3179.9091	3	3168.0000	3197.5227	113	0.
93%								
*	39	3	integral	0	3186.0000	3197.3990	126	0.
36%								
	40	3	3193.7500	1	3186.0000	3197.3990	128	0.
36%								

Cover cuts applied: 9
Zero-half cuts applied: 2
Gomory fractional cuts applied: 1

Solution pool: 5 solutions saved.
MIP-Integer optimal solution: Objective = 3.1860000000e+03
Solution time = 0.01 sec. Iterations = 131 Nodes = 44

In that example, ILOG CPLEX found the **optimal objective function value** of 3.186 after exploring 44 nodes and performing 131 (dual simplex) iterations, and ILOG CPLEX found an integral LP solution, designated by the star (*) and the word *integral* at node 39. That

log reports a **heuristic solution**, designated by a star (*) and a plus (+) after the node number, at the root and at node 20. 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** or **new incumbent**. 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 the objective function at the node is worse than the cutoff value for branch & cut; or if the linear programming relaxation at the node supplies an integer solution.) This column is left blank for lines that report that ILOG CPLEX found a **new incumbent** by primal heuristics. A plus (+) after the node number distinguishes such lines.

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 column titled *Best Integer* is blank; otherwise, it records the objective value of 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, the value is printed; otherwise, hyphens are printed. The gap is computed as:

$$\frac{(\text{best integer} - \text{best node}) * \text{objsen}}{(\text{abs}(\text{best integer}) + 1e-10)}$$

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. Moreover, if the populate procedure of the solution pool is invoked, the printed gap value may become negative after the optimal solution has been found and proven optimal.

ILOG CPLEX also logs its **addition of cuts to a model**. In the previous sample node log file, ILOG CPLEX reports that it added a variety of cuts (cover cuts, zero-half cuts, Gomory fractional cuts).

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 previous sample node log file shows that a heuristic found a solution at node 20. The + denotes an incumbent generated by the heuristic.

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 interval parameter, whichever is greater.)

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.

```
MIP-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 presents a typical MIP problem. Here is the node log file for that problem with the default setting of the MIP display parameter and one thread:

```
Tried aggregator 1 time.
Aggregator did 1 substitutions.
Reduced MIP has 2 rows, 3 columns, and 6 nonzeros.
Presolve time = 0.00 sec.
MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
Parallel mode: none, using 1 thread.
Root relaxation solution time = 0.00 sec.
```

Nodes		Cuts/					
Node	Left	Objective	IInf	Best Integer	Best Node	ItCnt	Gap
* 0+	0			34.0000	125.2083	3	268.26%
* 0+	0			122.5000	125.2083	3	2.21%
0	0	infeasible		122.5000	Fract: 1	3	0.00%

```
Solution pool: 2 solutions saved
MIP-Integer optimal solution: Objective = 1.2250000000e+02
Solution time = 0.00 sec. Iterations = 3 Nodes = 0
```

These additional items appear only in the node log file (not on screen) in conventional branch & cut:

- ◆ **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;
- NodeID specifies the node identifier.
- ◆ Parent specifies the NodeID of the parent.
- ◆ Depth indicates the depth of this node in the branch & cut tree.

Those additional items are not applicable in dynamic search.

Troubleshooting MIP performance problems

Describes symptoms of performance problems in MIP optimization and recommends remedies.

In this section

Introducing trouble shooting for MIP performance

Describes the background of MIP performance problems.

Too much time at node 0

Describes remedies for excessive time spent to solve root relaxation.

Trouble finding more than one feasible solution

Describes remedies for failure to find more than one feasible solution.

Large number of unhelpful cuts

Describes remedies for too much time spent to generate cuts.

Lack of movement in the best node

Describes conditions in which strong branching may help.

Time wasted on overly tight optimality criteria

Describes the effect of optimality tolerance on performance.

Slightly infeasible integer variables

Describes remedies for slightly infeasible integer variables.

Running out of memory

Describes remedies for limited memory.

Difficulty solving subproblems: overcoming degeneracy

Describes ways to overcome degeneracy as a source of difficulty in solving subproblems.

Unsatisfactory optimization of subproblems

Describes ways to overcome unsatisfactory optimization of difficult subproblems.

Introducing trouble shooting for MIP performance

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* 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* 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 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

If you observe that a very long time passes before the search 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 saying "Root relaxation solution time =" after 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*, *Solving problems with a quadratic objective (QP)*, and *Solving problems with quadratic constraints (QCP)* are applicable to tuning the root solve of a MIP model, too. In particular, it is worth considering setting the `RootAlg` parameter to a nondefault setting, such as the barrier optimizer, to see if a simple change in algorithm will speed up this step sufficiently.

For some problems, ILOG CPLEX will spend a significant amount of time performing computation at node 0, apart from solving the continuous LP, QP, or QCP root 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 and will eventually be proven 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 parameter *MIP heuristic frequency* (`HeurFreq`, `CPX_PARAM_HEURFREQ`) to a value such as 10, 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 parameter *MIP node selection strategy* (`NodeSel`, `CPX_PARAM_NODESEL`) 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.

The settings 1 (one) and 4 of the *MIP emphasis switch* (`MIPEmphasis`, `CPX_PARAM_MIPEMPHASIS`) both address the issue of finding feasible solutions. These parameter settings are also worth considering for this symptom of difficulty finding more than one feasible solution.

Solution polishing also helps you find additional solutions. A good strategy in this respect is to let branch & cut find the first feasible solution and then let solution polishing improve it. For instructions to apply this strategy, see *Finding a first solution to improve by polishing*.

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 selectively (that is, by type of cut) or completely.

To turn off cuts selectively, use the cut parameters summarized in *Parameters for controlling cuts*, with a value of -1 (minus one). For example, in the Interactive Optimizer, the following command turns off only cover cuts:

```
set mip cuts covers -1
```

To limit types of cuts generated, consider the *type of cut limit* parameter (EachCutLim, CPX_PARAM_EACHCUTLIM).

To limit the number of passes that ILOG CPLEX executes to generate cuts, consider the *number of cutting plane passes* parameter (CutPass, CPX_PARAM_CUTPASS).

To turn off all cuts, set the cut pass parameter to -1 (minus one). For example, in the Interactive Optimizer, use the following command to turn off all generation of all cuts:

```
set mip cuts all -1
```

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 variable selection parameter to a value of 3. For documentation of that parameter, see *MIP variable selection strategy* in the *ILOG CPLEX Parameters Reference Manual*.

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 nondefault 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 `ObjDiff` 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 `RelObjDiff` 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 ILOG CPLEX at default settings may contain solution values for the discrete variables that violate integrality by a small amount. The *integrality tolerance* parameter (`EpInt`, `CPX_PARAM_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, only consuming additional computing time.

However, some formulations combine discrete and continuous variables, for example, involving constraint coefficients over a million in magnitude, where even a small integrality violation can be magnified elsewhere in the model. In such situations, you may attempt to address this variation by tightening the simplex *feasibility tolerance* (`EpRHS`, `CPX_PARAM_EPRHS`) from its default value to its minimum; also tighten `EpInt` to a similar value, and re-run the MIP optimization from the beginning.

If this adjustment is insufficient to give satisfactory results, you can also try setting `EpInt` all the way to zero, preferably in conjunction with a tightened `EpRHS` setting. This very tight integrality tolerance directs ILOG CPLEX to attempt to branch on any integer infeasibility, no matter how small. Numeric round-off 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 `EpInt` or is above $1e-10$ when `EpInt` 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 round-off error after presolve uncrush, but extra care in using the solution may be advisable in numerically sensitive formulations.

If these suggestions are not appropriate for your problem, another alternative to consider is reformulation of your model with indicator constraints. *Indicator constraints in optimization* offers more information about that alternative.

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*.) 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*, 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 after 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*, 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.
MIP-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:

- ◆ *memory available for working storage*: `WorkMem` in Concert Technology or `CPX_PARAM_WORKMEM` in the Callable Library
- ◆ *node storage file switch*: `NodeFileInd` in Concert Technology or `CPX_PARAM_NODEFILEIND` in the Callable Library
- ◆ *tree memory limit*: `TreLim` in Concert Technology or `CPX_PARAM_TRELIM` in the Callable Library
- ◆ *directory for working files*: `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 decided by the setting of the *node storage file switch* (`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). *Parameter values for node file storage* summarizes these different options.

Parameter values for node file storage

Value	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 *directory for working files* 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 *tree memory limit* parameter (`TreLim`, `CPX_PARAM_TRELIM`) to limit the size of the tree so that it does not exceed available disk space, when you choose settings 2 or 3 in the *node storage file switch* (`NodeFileInd`, `CPX_PARAM_NODEFILEIND`). 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 *Node selection parameter settings for node search type*. 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, set the *MIP variable selection strategy* parameter (`VarSel`, `CPX_PARAM_VARSSEL`) to the value 3. Strong branching requires substantial computational effort at each node to decide 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 optimization of subproblems*. You may also consider a stronger *dual simplex pricing algorithm*, such as dual steepest-edge pricing (that is, the parameter `DPriInd` or `CPX_PARAM_DPRIIND` set to the value 2).

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

A different option is to solve as few subproblems of the original model as possible by switching to solution polishing after you find a first feasible solution. This strategy is appropriate if obtaining good integer solutions is more important than obtaining a proof of optimality. For details about how to apply solution polishing, see *Solution polishing*.

Unsatisfactory optimization of subproblems

In some problems, you can improve performance by evaluating how the continuous LP or QP subproblems are solved at the nodes in the search space, and then possibly modifying the choice of algorithm to solve them.

QCP subproblems are solved only by the barrier optimizer. However, MIQCP models are not always solved by a sequence of QCP subproblems. The *MIQCP strategy switch* (MIQCPStrat, CPX_PARAM_MIQCPSTRAT) allows you to control what kinds of subproblems are solved in a mixed integer quadratically constrained programming model. Consequently, the following suggestions may also help that class of problem as well.

You can control which algorithm ILOG CPLEX applies to the root 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 and difficult subproblems

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 suggests that another optimizer could be more efficient. *Parameter settings for RootAlg and NodeAlg* summarizes the values available for the RootAlg parameter.

To set the *MIP starting algorithm* parameter:

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

Parameter settings for RootAlg and NodeAlg

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)

Concert Technology Enumeration	Callable Library Symbolic Constant	Setting	Calls this Optimizer
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 and difficult subproblems

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 *Parameter settings for RootAlg and NodeAlg*.

To set the *MIP subproblem algorithm* 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).

Solution polishing and difficult subproblems

When subproblems are not solved satisfactorily, another option is to solve as few subproblems of the original model as possible by switching to solution polishing as soon as a first feasible

solution is found. This strategy is helpful when finding a good integer solution is more important than proving optimality. For more information about this strategy, see *Solution polishing* in this manual.

Examples: optimizing a simple MIP problem

Introduces samples that demonstrate how to optimize a MIP with the ILOG CPLEX Component Libraries.

In this section

ilomipex1.cpp

Shows how to optimize a MIP model in the C++ API.

MIPex1.java

Shows how to optimize a MIP model in the Java API.

MIPex1.cs and MIPex1.vb

Shows how to optimize a MIP model in the .NET API.

mipex1.c

Shows how to optimize a MIP model in the C API.

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

MIPex1.java

Also available among the examples distributed with the product is a Java implementation of the same MIP.

MIPex1.cs and MIPex1.vb

Also available among the examples distributed with the product are a C#.NET and a Visual Basic.NET implementation of the same MIP.

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*.
- ◆ The routine `setproblemdata` has an argument, `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 `CPXchgprobtype` to change the problem type to `CPXchgprobtype`.
- 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

Introduces samples that show to solve a MIP with the Component Libraries when the problem data is stored in a file.

In this section

ilomipex2.cpp

Shows how to read data from a file and solve a MIP model in the C++ API.

mipex2.c

Shows how to read data from a file and solve a MIP model in the C API.

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 `env`, an instance of `IloEnv`, to initialize the Concert Technology environment and `IloModel model(env)` to create a problem object. Before the application 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 specify 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.

Solution pool: generating and keeping multiple solutions

Introduces the solution pool for storing multiple solutions to a mixed integer programming problem (MIP) and explains techniques for generating and managing those solutions.

In this section

What is the solution pool?

Defines the solution pool for storing multiple solutions of a MIP model.

Example: simple facility location problem

Describes a model used in documentation of the solution pool.

Filling the solution pool

Describes ways to fill the solution pool.

Accumulating incumbents in the solution pool

Describes accumulation of incumbents in the solution pool.

Populating the solution pool

Explains populate, the fundamental procedure of the solution pool.

Choosing whether to accumulate or populate

Explains differences between accumulating incumbents and generating multiple solutions in the solution pool.

Enumerating all solutions

Describes difficulties of enumerating all solutions.

Impact of change on the solution pool

Describes changes to a model or its context that may have an impact on the solution pool and the solutions stored there.

Examining the solution pool

Describes access to generic information about the solution pool.

Accessing a solution in the solution pool

Describes access to a particular solution in the solution pool.

Using solutions from the solution pool

Describes routines and methods to exploit solutions from the solution pool.

Deleting solutions from the solution pool

Describes routines and methods to delete solutions from the solution pool.

The incumbent and the solution pool

Describes access to the incumbent from the solution pool.

Parameters of the solution pool

Describes parameters to control the solution pool.

Filtering the solution pool

Documents filters of the solution pool. Filters offer a means of controlling properties of the solutions you generate and store.

What is the solution pool?

The *solution pool* stores multiple solutions to a mixed integer programming (MIP) model. With this feature, you can direct the algorithm to generate multiple solutions in addition to the optimal solution.

For example, some constraints may be difficult to formulate efficiently as linear expressions, or the objective may be difficult to quantify exactly. In such cases, obtaining multiple solutions will help you choose one which best fits all your criteria, including the criteria that could not be expressed easily in a conventional MIP model.

Furthermore, you can use the solution pool and tools associated with it to explore and evaluate alternative solutions in a variety of ways:

- ◆ You can collect solutions within a given percentage of the optimal solution. To do so, apply the solution pool gap parameters (relative or absolute), as explained in *Parameters of the solution pool*.
- ◆ You can collect a set of diverse solutions. To do so, use the *solution pool replacement strategy* parameter (`SolnPoolReplace`, `CPX_PARAM_SOLNPOOLREPLACE`) to set the solution pool replacement strategy to `CPX_SOLNPOOL_DIV`, as explained in the documentation of that parameter in the *ILOG CPLEX Parameter Reference Manual*. In order to control the diversity of solutions even more finely, apply a diversity filter, as explained in *Diversity filters*.
- ◆ In an advanced application of this feature, you can collect solutions with specific properties. To do so, see *Filtering the solution pool*.
- ◆ You can collect all solutions or all optimal solutions to a MIP model. To do so, set the *solution pool intensity* parameter (`SolnPoolIntensity`, `CPX_PARAM_SOLNPOOLINTENSITY`) to its highest value.

Tip: The solution pool distinguishes solutions by the values of their discrete variables only. For more explanation of this point, see *Limitations due to continuous variables and finite precision*

Example: simple facility location problem

A simple version of a facility location problem appears throughout this sequence of topics to show how the solution pool and the tools associated with it work. Here's a description of the problem: a company is considering opening as many as four warehouses in order to serve nine different regions. The goal is to minimize the sum of fixed costs associated with opening warehouses (constraint $c2$) as well as the various transportation costs incurred to ship goods from the warehouses to the regions (constraint $c3$).

Whether or not to open a warehouse is represented by binary variable x_i , for $i=1$ to 4.

Whether or not to ship goods from warehouse i to region j is represented by binary variable y_{ji} , for $j=1$ to 9 and $i=1$ to 4.

Each region needs a specified amount of goods, and each warehouse can store only a limited quantity of goods (constraints $c4$ to $c7$). In addition, each region must be served by exactly one warehouse (constraints $c8$ to $c16$). Constraints $c17$ to $c52$ complete the model by stating that warehouse i must be open in order for goods to be shipped from warehouse i to any region j .

The model for this simple facility location problem is available online in the formatted LP file `yourCPLEXdir/examples/data/location.lp`. In standard form, a model for the simple facility location problem looks like this:

```
Minimize
  obj: cost
Subject To
  c1: - cost + fixed + transport = 0
  c2: - fixed + 130 x1 + 150 x2 + 170 x3 + 180 x4 = 0
  c3: - transport
      + 10 y11 + 30 y12 + 25 y13 + 55 y14
      + 10 y21 + 25 y22 + 25 y23 + 45 y24
      + 20 y31 + 23 y32 + 30 y33 + 40 y34
      + 25 y41 + 10 y42 + 26 y43 + 40 y44
      + 28 y51 + 12 y52 + 20 y53 + 29 y54
      + 36 y61 + 19 y62 + 16 y63 + 22 y64
      + 40 y71 + 39 y72 + 22 y73 + 27 y74
      + 75 y81 + 65 y82 + 55 y83 + 35 y84
      + 34 y91 + 43 y92 + 41 y93 + 62 y94 = 0

  c4: 10 y11 + 10 y21 + 12 y31 + 15 y41 + 15 y51 + 15 y61 + 20 y71 + 25 y81 +
30 y91 - 90 x1 <= 0
  c5: 10 y12 + 10 y22 + 12 y32 + 15 y42 + 15 y52 + 15 y62 + 20 y72 + 25 y82 +
30 y92 - 110 x2 <= 0
  c6: 10 y13 + 10 y23 + 12 y33 + 15 y43 + 15 y53 + 15 y63 + 20 y73 + 25 y83 +
30 y93 - 130 x3 <= 0
  c7: 10 y14 + 10 y24 + 12 y34 + 15 y44 + 15 y54 + 15 y64 + 20 y74 + 25 y84 +
```

```

30 y94 - 150 x4 <= 0
c8: y11 + y12 + y13 + y14 = 1
c9: y21 + y22 + y23 + y24 = 1
c10: y31 + y32 + y33 + y34 = 1
c11: y41 + y42 + y43 + y44 = 1
c12: y51 + y52 + y53 + y54 = 1
c13: y61 + y62 + y63 + y64 = 1
c14: y71 + y72 + y73 + y74 = 1
c15: y81 + y82 + y83 + y84 = 1
c16: y91 + y92 + y93 + y94 = 1

```

```

c17: x1 - y11 >= 0
c18: x1 - y21 >= 0
c19: x1 - y31 >= 0
c20: x1 - y41 >= 0
c21: x1 - y51 >= 0
c22: x1 - y61 >= 0
c23: x1 - y71 >= 0
c24: x1 - y81 >= 0
c25: x1 - y91 >= 0
c26: x2 - y12 >= 0
c27: x2 - y22 >= 0
c28: x2 - y32 >= 0
c29: x2 - y42 >= 0
c30: x2 - y52 >= 0
c31: x2 - y62 >= 0
c32: x2 - y72 >= 0
c33: x2 - y82 >= 0
c34: x2 - y92 >= 0
c35: x3 - y13 >= 0
c36: x3 - y23 >= 0
c37: x3 - y33 >= 0
c38: x3 - y43 >= 0
c39: x3 - y53 >= 0
c40: x3 - y63 >= 0
c41: x3 - y73 >= 0
c42: x3 - y83 >= 0
c43: x3 - y93 >= 0
c44: x4 - y14 >= 0
c45: x4 - y24 >= 0
c46: x4 - y34 >= 0
c47: x4 - y44 >= 0
c48: x4 - y54 >= 0
c49: x4 - y64 >= 0
c50: x4 - y74 >= 0
c51: x4 - y84 >= 0
c52: x4 - y94 >= 0

```

Binaries

```

x1 x2 x3 x4
y11 y12 y13 y14 y21 y22 y23 y24 y31 y32 y33 y34
y41 y42 y43 y44 y51 y52 y53 y54 y61 y62 y63 y64
y71 y72 y73 y74 y81 y82 y83 y84 y91 y92 y93 y94

```

Filling the solution pool

There are two ways to fill the solution pool associated with a model:

- ◆ You can accumulate successive incumbents, as explained in *Accumulating incumbents in the solution pool*.
- ◆ You can generate alternative solutions, as explained in *Populating the solution pool*.

The difference between those ways is explained in *Choosing whether to accumulate or populate*.

Other details about filling and managing the solution pool are explained in *Deleting solutions from the solution pool* and *Model changes and the solution pool*.

Accumulating incumbents in the solution pool

MIP optimization automatically adds incumbents to the solution pool as they are discovered when you call it in one of these conventional ways:

- ◆ In Concert Technology, you invoke MIP optimization by means of one of these methods:
 - `IloCplex::solve` in the C++ API.
 - `IloCplex.solve` in the Java API.
 - `Cplex.Solve` in the .NET API.
- ◆ In the Callable Library (C API), you invoke the MIP optimizer by means of the routine `CPXmipopt`.
- ◆ In the Interactive Optimizer, you invoke the command `mipopt`.

For example, if you read the model of *Example: simple facility location problem* into the Interactive Optimizer and invoke the usual command `mipopt`, MIP optimization finds solutions that it stores in the solution pool, and the log looks something like this (allowing for variation in the MIP path):

```
Solution pool: 4 solutions saved.
```

```
MIP - Integer optimal solution: Objective = 4.9900000000e+02  
Solution time = 0.12 sec. Iterations = 197 Nodes = 33
```


Populating the solution pool

Explains populate, the fundamental procedure of the solution pool.

In this section

What is populating the solution pool?

Introduces the populate procedure of the solution pool.

Invoking the populate procedure

Describes how to invoke the populate procedure of the solution pool.

Algorithm of the populate procedure

Describes the algorithm of the populate procedure.

Example: calling populate

Shows an example of the populate procedure in the Interactive Optimizer.

Stopping criteria for the populate procedure

Defines stopping criteria of the populate procedure.

What is populating the solution pool?

ILOG CPLEX also provides a procedure specifically to generate multiple solutions. You can invoke this procedure either as an alternative to the usual MIP optimizer or as a successor to the MIP optimizer. You can also invoke this procedure many times in a row in order to explore the solution space differently. In particular, you may invoke this procedure multiple times to find additional solutions, especially if the first solutions found are not satisfactory.

The following topics tell you more about the populate procedure.

Invoking the populate procedure

The populate procedure is available in all application programming interfaces (APIs) of ILOG CPLEX.

- ◆ In Concert Technology, you invoke this procedure by means of the method:
 - `IloCplex::populate` in the C++ API.
 - `IloCplex.populate` in the Java API.
 - `Cplex.Populate` in the .NET API.
- ◆ In the Callable Library (C API), you invoke this procedure by means of the routine `CPXpopulate`.
- ◆ In the Interactive Optimizer, you invoke this procedure by means of the command `populate`.

For examples of populating the solution pool in an application, see *yourCPLEXinstallation/examples/src/populate.c* and *ilopopulate.cpp* as well as *Populate.java* and *Populate.cs*.

Algorithm of the populate procedure

Briefly, the algorithm that populates the solution pool works in two phases.

In the first phase, it solves the model to optimality (or some stopping criterion set by the user) while it sets up a search tree for the second phase.

In the second phase, it generates multiple solutions by using the information computed and stored in the first phase and by continuing to explore the tree.

The amount of preparation in the first phase and the intensity of exploration in the second phase are controlled by the *solution pool intensity* parameter:

- ◆ `SolnPoolIntensity` in Concert Technology;
- ◆ `CPX_PARAM_SOLNPOOLINTENSITY` in the Callable Library;
- ◆ `mip pool intensity` in the Interactive Optimizer.

After a model has been read (or created), the first call to populate will carry out both the first and second phase. In the general case, subsequent calls to populate will re-use stored information and proceed with the continuation of the second phase. The first phase will be re-computed if:

- ◆ the value of the pool intensity parameter has increased between successive calls of populate;
- ◆ any filters have been deleted.

The details of the algorithm that populates the solution pool are published in the paper titled "Generating Multiple Solutions for Mixed Integer Programming Problems," by Emilie Danna, Mary Fenelon, Zhonghao Gu, and Roland Wunderling, in the **Proceedings of the Twelfth Conference on Integer Programming and Combinatorial Optimization (IPCO 2007)**, LNCS 4513, pages 280 - 294.

Example: calling populate

You can generate multiple solutions with populate. To see this effect in the **Interactive Optimizer**, first read the example cited in *Example: simple facility location problem*, like this:

```
read location.lp
populate
```

At default settings in the Interactive Optimizer, you will see results such as these:

```
Populate: phase I
Tried aggregator 1 time.
MIP Presolve eliminated 3 rows and 3 columns.
MIP Presolve modified 47 coefficients.
Reduced MIP has 49 rows, 40 columns, and 148 nonzeros.
Presolve time = 0.01 sec.
Clique table members: 45.
MIP emphasis: balance optimality and feasibility.
Root relaxation solution time = 0.04 sec.
```

	Node	Nodes Left	Objective	IInf	Best Integer	Cuts/ Best Node	ItCnt	Gap
	0	0	452.1107	27		452.1107	51	
*	0+	0			549.0000	452.1107	51	17.
65%	0	0	468.2224	23	549.0000	Cuts: 17	64	14.
71%	0	0			512.0000	468.2224	64	8.
*	0+	0			512.0000	468.2224	64	8.
55%	0	0	470.5942	23	512.0000	Cuts: 2	68	8.
09%	0	0	470.6800	20	512.0000	Cuts: 3	70	8.
07%	0	0	470.6800	20	512.0000	470.6800	70	8.
07%	0	2	470.6800	20	512.0000	470.6800	70	8.
*	10	6	integral	0	499.0000	479.9271	129	3.
82%								

```
Cover cuts applied: 2
Zero-half cuts applied: 2
Gomory fractional cuts applied: 1
```

```
Populate: phase II
```

```

MIP emphasis: balance optimality and feasibility.
  100   26   infeasible           499.0000       500.0000       234   -0.
20%

Cover cuts applied:  2
Zero-half cuts applied:  2
Gomory fractional cuts applied:  1

```

```

Solution pool: 20 solutions saved.

```

```

Populate - Populate solution limit exceeded, integer optimal:  Objective =
4.99000000000e+02
Solution time =      0.54 sec.  Iterations = 261  Nodes = 193 (34)

```

In that log, you see that the procedure executed its first and second phases. It reports parameter settings, such as MIP emphasis, like other optimization logs. It also reports how many solutions it found. It stops when it reaches the populate limit (20 solutions in this example).

Interestingly, the gap printed in that log becomes negative in the second phase of populate. At the end of the first phase of populate, the model was solved to optimality; the best node value and the best integer value coincided and were equal to the optimal objective value; the gap was zero. Early in the second phase, the best integer value remained equal to the optimal objective value, but as populate progressed, nodes were explored and fathomed. At some point, all nodes with a relaxation value equal to the optimal objective value were fathomed. This fathoming explains why the best node value increased above the optimal objective value (for a minimization problem, such as this example) as the search space was explored in the second phase. Recall that the gap value is computed as:

$$(\text{bestInteger} - \text{bestNode}) * \text{objSense} / (\text{abs}(\text{bestInteger}) + 1e-10)$$

Consequently, the gap can become negative. A negative gap value ($-g\%$) indicates that the search space explored by populate does not contain any more solutions that are less than $g\%$ worse than the optimal objective value.

You can invoke the populate procedure multiple times. In successive invocations, it will re-use information it has accumulated in previous invocations. For example, if you then immediately invoke populate a second time on this model, it re-uses the information it gathered in the previous invocation to resume its second phase, like this:

```

CPLEX> populate

Populate: phase II
MIP emphasis: balance optimality and feasibility.
  200   32   infeasible           499.0000       512.0000       268   -2.
61%
  300   38   infeasible           499.0000       514.0000       282   -3.
01%
  400   44     516.0000           1           499.0000       516.0000       295   -3.

```

```
41%      500      48      518.0000      1      499.0000      518.0000      312      -3.
81%

Cover cuts applied:  2
Zero-half cuts applied:  2
Gomory fractional cuts applied:  1

Solution pool: 40 solutions saved.

Populate - Populate solution limit exceeded, integer optimal:  Objective =
4.99000000000e+02
Solution time =      0.23 sec.  Iterations = 320  Nodes = 532 (53)
```

In this second invocation, `populate` does not disturb the twenty solutions already accumulated in the solution pool, and it continues to search for another twenty solutions before stopping at its default limit again.

The status line of both invocations of `populate` indicates that the optimal solution of the model has been found. Nevertheless, `populate` continues to produce solutions: optimality is not the stopping criterion for populating the solution pool. For more detail about stopping criteria, see *Stopping criteria for the populate procedure*.

Stopping criteria for the populate procedure

Optimality is not a stopping criterion for the populate procedure. Even if the optimality gap is zero, this procedure will still try to find alternative solutions. The stopping criteria for populating the solution pool are these:

- ◆ **Populate limit.** This parameter controls how many solutions are generated before the populate procedure stops. Its default value is 20. Consequently, the procedure stopped after generating 20 solutions in the example with model `location.lp` in *Example: calling populate*.

limit on number of solutions generated for solution pool

- `PopulateLim` in Concert Technology
- `CPX_PARAM_POPULATELIM` in the Callable Library (C API)
- `mip limits populate` in the Interactive Optimizer

Note: The parameter to limit the number of integer solutions in a MIP (*MIP integer solution limit*: `IntSolLim` in Concert Technology, `CPX_PARAM_INTSOLLIM` in the Callable Library, or `mip limits solutions` in the Interactive Optimizer) does not apply to the populate procedure; if you want to limit the populate procedure, apply the populate limit parameter (`PopulateLim`, `CPX_PARAM_POPULATELIM`) instead.

- ◆ *optimizer time limit*, as in a standard MIP optimization.
 - `TiLim` in Concert Technology
 - `CPX_PARAM_TILIM` in the Callable Library (C API)
 - `timelimit` in the Interactive Optimizer
- ◆ *MIP node limit*, as in a standard MIP optimization.
 - `NodeLim` in Concert Technology
 - `CPX_PARAM_NODELIM` in the Callable Library (C API)
 - `mip limit nodes` in the Interactive Optimizer
- ◆ In the absence of other stopping criteria, the populate procedure stops when it cannot enumerate any more solutions.

In particular, if you specify an objective tolerance with the relative or absolute solution pool gap parameters, populate stops if it cannot enumerate any more solutions within the specified objective tolerance.

However, there may exist additional solutions that are feasible, and if you have specified an objective tolerance, those feasible solutions may also satisfy this additional criterion. Depending on the solution pool intensity parameter, populate may or may not enumerate all possible solutions. Consequently, populate may stop when it has enumerated only a subset of the solutions satisfying your criteria.

Choosing whether to accumulate or populate

Explains differences between accumulating incumbents and generating multiple solutions in the solution pool.

In this section

What's the difference between accumulating and populating?

Contrasts the populate procedure with MIP optimization.

Advanced use: interaction of MIP optimization and populate

Explains when to invoke the populate procedure.

Example: using populate after MIP optimization

Illustrates contrast between MIP optimization and the populate procedure.

What's the difference between accumulating and populating?

Both MIP optimization and populate generate a series of solutions, but the two procedures differ in their aims. The aim of MIP optimization is optimality: after a solution has been found, MIP optimization will generate only solutions of improving objective value, and the procedure will stop when optimality has been proven. In contrast, the aim of populate is to generate as many solutions as possible: after a solution has been found, populate may generate solutions of both improving and degrading objective value, and it will stop only when it cannot generate any additional solutions or because other stopping criteria intervene.

In order to decide which procedure is better for your application, you should first try the MIP optimizer. If the solutions produced are sufficient for your application, then the MIP optimizer is the appropriate choice. If not, then you should try populate to generate more solutions and to have more control over the properties of the generated solutions.

Advanced use: interaction of MIP optimization and populate

Should you call MIP optimization and then populate, or should you call populate alone?

You can call populate after you call the MIP optimizer, or you can call populate on its own after you read or create a model. In order to decide which to do, you need to know more about the two procedures.

Recall that the algorithm underlying populate works in two phases. If you call the MIP optimizer after the model is read, it will gather and store information about the search as it solves the model. In practice, its activity constitutes the first phase of the populate algorithm. In the general case, if you then call populate, populate will re-use the information stored by the MIP optimizer and carry out only the second phase.

In contrast, if you call populate immediately after the model is read, populate will perform both the first phase and the second phase.

If you specify a nondefault setting of the pool intensity parameter, then calling the MIP optimizer and afterwards calling populate will give the same results in terms of performance and solutions generated as calling populate alone. (The exception to this generalization occurs when the pool intensity parameter is set at its default value, 0 (zero) that is, automatic. For details about that case, see the documentation of the solution pool intensity parameter.)

Calling populate alone is simpler than calling populate after MIP optimization. However, if you want more control over the details of the two phases (for example, if you want to specify different stopping criteria for each phase), then you need to call MIP optimization followed by populate, instead of calling populate alone. The risk associated with this approach is that populate might not be able to reuse the information about the tree from the previous MIP optimization; in that case, populate will start from scratch; that is, it again performs the first phase, followed by the second phase. In particular, this repetition of the first phase will happen if you increase the pool intensity parameter between the call to MIP optimization and the call to populate.

More information about this topic can be found in the documentation about the *solution pool intensity* parameter (`SolnPoolIntensity`, `CPX_PARAM_SOLNPOOLINTENSITY`) in the *ILOG CPLEX Parameter Reference Manual*.

In short, if you want the simplicity of a black box, call populate alone; if you need more control, call MIP optimization, then populate.

Example: using populate after MIP optimization

After invoking MIP Optimization, you can generate additional solutions with populate. You can use this possibility to get a few additional solutions quickly if the solutions obtained during MIP Optimization are not satisfactory. However, as explained in *Advanced use: interaction of MIP optimization and populate*, the sequence MIP Optimization followed by populate is especially useful to control the parameters and the stopping criteria of each phase of populate.

Consider again the model in *Example: simple facility location problem*. Suppose that the transportations costs are subject to fluctuations, and consequently it does not make sense to spend time optimizing the model exactly to optimality. You can set the MIP gap tolerance (*absolute MIP gap tolerance*: EpAGap, CPX_PARAM_EPAGAP; *relative MIP gap tolerance* EpGap, CPX_PARAM_EPGAP) to a value higher than the default (in this example: 5%) so that the MIP optimization, which constitutes the first phase of populate, stops earlier. Then, populate will go immediately into the second phase, so it can start producing solutions sooner.

The commands to reproduce this situation looks like this in the Interactive Optimizer:

```
read location.lp
set mip pool intensity 2
set mip tolerances mipgap 0.05
mipopt
populate
```

MIP optimization (as executed by mipopt in the Interactive Optimizer) shows these results:

```
Solution pool: 3 solutions saved.

MIP - Integer optimal, tolerance (0.05/1e-06): Objective = 4.9900000000e+02
Current MIP best bound = 4.7976250000e+02 (gap = 19.2375, 3.86%)
Solution time = 0.05 sec. Iterations = 135 Nodes = 11 (6)
```

Populate (following mipopt in the Interactive Optimizer) shows results like these:

```
Solution pool: 23 solutions saved.

Populate - Populate solution limit exceeded, integer feasible:
Objective = 4.9900000000e+02
Current MIP best bound = 4.9278787879e+02 (gap = 6.21212, 1.24%)
Solution time = 0.05 sec. Iterations = 271 Nodes = 261 (53)
```

In this example, the *solution pool intensity* (SolnPoolIntensity, CPX_PARAM_SOLNPOOLINTENSITY) is set to 2 because the default automatic value of this parameter for the sequence MIP optimization followed by populate is not the fastest

possible setting for generating a large number of solutions. If you use this sequence of commands to control precisely the behavior of the optimizer in the first and second phase of populate, it is a good idea to reset the pool intensity parameter yourself, rather than relying on its default value.

Enumerating all solutions

Describes difficulties of enumerating all solutions.

In this section

How to enumerate all solutions

Tells how to enumerate all solutions of a MIP model.

Limitations due to continuous variables and finite precision

Describes limitations due to finite precision arithmetic.

Limitations due to unbounded MIP models

Describes limitations due to unbounded MIP models.

Limitations due to numeric difficulties

Describes limitations due to numeric difficulties.

How to enumerate all solutions

You can also enumerate all solutions that are valid for a specific criterion. For example, if you want to enumerate all alternative optimal solutions, do the following:

1. Set the *absolute gap for solution pool* parameter (SolnPoolAGap, CPX_PARAM_SOLNPOOLAGAP) to 0.0 (zero).
2. Set the *solution pool intensity* parameter to 4 (SolnPoolIntensity, CPX_PARAM_SOLNPOOLINTENSITY).
3. Set the *limit on number of solutions generated for solution pool* (PopulateLim, CPX_PARAM_POPULATELIM) to a value sufficiently large for your model; for example, 2 100 000 000.
4. Call populate.

Beware, however, that, even for small models, the number of possible solutions is likely to be huge. Consequently, enumerating all of them will take time and consume a large quantity of memory.

In addition, when you attempt to enumerate all solutions, some restrictions apply, as explained in the following sections.

- ◆ *Limitations due to continuous variables and finite precision*
- ◆ *Limitations due to unbounded MIP models*
- ◆ *Limitations due to numeric difficulties*

Limitations due to continuous variables and finite precision

There may be an infinite number of possible values for a continuous variable, and it is not practical to enumerate all of them on a finite-precision computer. Therefore, populate gives only one solution for each set of discrete variables, even though there may exist several solutions that have the same values for all discrete variables but different values for continuous variables.

Limitations due to unbounded MIP models

Likewise, for the same reason, populate does not generate all possible solutions for unbounded MIP models. As soon as the proof of unboundness is obtained, populate stops.

Limitations due to numeric difficulties

ILOG CPLEX uses numerical methods of finite-precision arithmetic. Consequently, the feasibility of a solution depends on the value given to tolerances. Two parameters define the tolerances that assess the feasibility of a MIP solution:

- ◆ the *integrality tolerance* (E_{pInt} , CPX_PARAM_EPINT);
- ◆ the *feasibility tolerance* (E_{pRHS} , CPX_PARAM_EPRHS).

A solution may be considered feasible for one pair of values for these two parameters, and infeasible for a different pair. This phenomenon is especially noticeable in models with numeric difficulties, for example, in models with Big M coefficients.

Since the definition of a feasible MIP solution is subject to tolerances, the total number of solutions to a MIP model may vary, depending on the approach used to enumerate solutions, and on precisely which tolerances are used. In most models, this tolerance issue is not problematic for ILOG CPLEX. But, in the presence of numeric difficulties, ILOG CPLEX may create solutions that are slightly infeasible or integer infeasible, and therefore create more solutions than expected.

You can find more details about the topic of numeric difficulties in the *ILOG CPLEX User's Manual* in *Numeric difficulties* and *Slightly infeasible integer variables*.

Impact of change on the solution pool

Describes changes to a model or its context that may have an impact on the solution pool and the solutions stored there.

In this section

Changes between MIP optimization and populate

Describes changes to a MIP model between optimization and populate.

Persistence of solutions in the solution pool

Describes conditions under which solutions persist in the solution pool.

Model changes and the solution pool

Describes model changes and their impact on feasibility of solutions in the solution pool.

Changes between MIP optimization and populate

What might users do between a call of MIP optimization and a call of populate or between successive calls of populate?

Users can continue to call populate until they have a pool of solutions they are satisfied with. Between calls, users may examine solutions. If the solutions are satisfactory, users can stop calling populate. If the solutions are not satisfactory, then users can make changes to improve the solution pool. Changes that have an impact on the solutions that populate generates and stores include these:

- ◆ changing parameter settings;
- ◆ adding filters;
- ◆ removing filters;
- ◆ changing characteristics of filters.

Changes of the model itself, such as altering the objective function, are treated differently. For details, see *Model changes and the solution pool*.

Persistence of solutions in the solution pool

Successive calls to MIP optimization or populate create solutions that are stored in the solution pool. Each call to MIP optimization or populate applies to all solutions in the pool. In particular, CPLEX may replace solutions in the pool obtained during previous invocations of MIP optimization or populate if the pool is at its capacity and CPLEX finds new solutions satisfying the replacement criteria.

If you want to keep all solutions produced through all calls to MIP optimization or populate, then you must query the solution pool before calling MIP optimization or populate again and store the solutions in user-defined arrays.

Model changes and the solution pool

When a user modifies a model, for example, by adding constraints or changing the coefficients of the objective function, the existing solutions already in the solution pool may or may not be feasible in terms of the changed model. Therefore, immediately after changing a model, the user cannot access the solution pool. In fact, if the user attempts to query the solution pool after changing a model, the solution pool query routines and methods return an error: `CPXERR_NO_SOLNPOOL`.

However, the MIP starts constructed from the solutions in the solution pool before the changes to the model may still exist if those MIP starts have not been modified or deleted. If those unmodified MIP starts still exist, they are accessible through these methods and routines:

- ◆ `writeMIPStart` in the C++ API;
- ◆ `writeMIPStart(String)` in the Java API;
- ◆ `Cplex.WriteMIPStart` in the .NET API;
- ◆ `CPXgetmipstarts` and `CPXwritemipstarts` in the Callable Library (C API).

Moreover, if the *advanced start switch* (`AdvInd`, `CPX_PARAM_ADVIND`) is set to a value greater than zero, then after the user changes a model, the next call to MIP optimization or `populate` tries to process MIP starts corresponding to members of the solution pool to derive solutions for the newly changed model.

Examining the solution pool

In the Interactive Optimizer, the command `display solution pool` produces a synopsis about the solution pool.

Also in the Interactive Optimizer, the command `display solution list` shows the objective value of solutions in the pool, along with the percentage of discrete variables that take a value different from the incumbent. To display all solutions in the pool, use `display solution list *`. Alternatively, you can specify the indices of the solutions to display, for example: `display solution list 2-4`.

The information displayed by both of these commands in the Interactive Optimizer is available through methods or routines in Concert Technology and the Callable Library, as shown in *Accessing information about the solution pool*.

Accessing information about the solution pool summarizes methods, routines, and commands that access aggregated information about the solution pool itself, such information as the number of solutions in the pool, the number of solutions that have been replaced, and the arithmetic mean of the objective value of solutions in the pool.

Solutions are *replaced*, according to the replacement policy, when the pool reaches the limit on its capacity. In the presence of an objective tolerance specific to the solution pool, as specified either by the *relative gap for solution pool* parameter (`SolnPoolGap`, `CPX_PARAM_SOLNPOOLGAP`) or by the *absolute gap for solution pool* parameter (`SolnPoolAGap`, `CPX_PARAM_SOLNPOOLAGAP`), solutions that are generated but then deleted because of the improvement in the best integer value are also counted as replaced.

Accessing information about the solution pool

Purpose	Concert Technology	Callable Library	Interactive Optimizer
Number of solutions	<code>getSolnPoolNsolns</code>	<code>CPXgetsolnpoolnumsolns</code>	<code>display solution</code>
Number replaced	<code>getSolnPoolNreplaced</code>	<code>CPXgetsolnpoolnumreplaced</code>	<code>display solution</code>
Arithmetic mean of objective values	<code>getSolnPoolMeanObjValue</code>	<code>CPXgetsolnpoolmeanobjval</code>	<code>display solution</code>

Accessing a solution in the solution pool

If you want to examine all the solutions available in the solution pool, your application should loop from 0 (zero) to N-1 (that is, one less than the number of solutions in the pool).

To learn the number of solutions in the pool for such a loop, use one of the following methods or routines.

- ◆ In Concert Technology,
 - In the C++ API, use the method `IloCplex::getSolnPoolNsolns`.
 - In the Java API, use the method `IloCplex.getSolnPoolNsolns`.
 - In the .NET API, use the method `Cplex.GetSolnPoolNsolns`.
- ◆ In the Callable Library, use the routine `CPXgetsolnpoolnumsolns`.

Accessing solution information in the solution pool summarizes the methods, routines, or commands that access information about a given solution in the solution pool.

Accessing solution information in the solution pool

Purpose	Concert Technology	Callable Library	Interactive Optimizer
Objective value	<code>getObjValue</code>	<code>CPXgetsolnpoolobjval</code>	<code>display solution member i obj</code>
Value of variable	<code>getValues</code>	<code>CPXgetsolnpoolx</code>	<code>display solution member i var</code>
Slack in linear constraints	<code>getSlacks</code>	<code>CPXgetsolnpoolslack</code>	<code>display solution member i slacks</code>
Slack in quadratic constraints	<code>getSlacks</code>	<code>CPXgetsolnpoolqconstrslack</code>	<code>display solution member i qcslacks</code>
Quality	<code>getQuality</code>	<code>CPXgetsolnpoolintquality</code> or <code>CPXgetsolnpooldblquality</code>	<code>display solution member i quality</code>
Difference between solutions	(see Notes)	(see Notes)	<code>display solution difference i j</code>

Note: In the Interactive Optimizer, the command

```
display solution difference 1 2
```

compares the first and second solution. Likewise, the command

```
display solution difference 0 2
```

compares the incumbent and the second solution.

There is no exact equivalent of this `difference` command in Concert Technology or the Callable Library. In those APIs, first access the solution vector (for example, in the C++ API by means of the method `getValues` or in the C API by means of the routine `CPXgetsolnpoolx`) and then write your own comparison.

For a sample of these methods or routines, see the example in *yourCPLEXhome* /examples :

- ◆ `ilopopulate.cpp`
- ◆ `Populate.java`
- ◆ `Populate.cs` or `Populate.vb`
- ◆ `populate.c`

Using solutions from the solution pool

The solutions in the solution pool are available for use in applications or further optimizations. For example, you can write a particular solution from the solution pool to a solution file in SOL format.

- ◆ In Concert Technology
 - In the C++ API, use `IloCplex::writeSolution`.
 - In the Java API, use `IloCplex.writeSolution`.
 - In the .NET API, use `Cplex.WriteSolution`.
- ◆ In the Callable library, use the routine `CPXsolwritessolnpool`.
- ◆ In the Interactive Optimizer, use this command, where *i* represents the index of the solution in the solution pool: `write filename .sol i`

You can also write all the solutions from the solution pool into a single SOL file.

- ◆ In Concert Technology
 - In the C++ API, use `IloCplex::writeSolutions`.
 - In the Java API, use `IloCplex.writeSolutions`.
 - In the .NET API, use `Cplex.WriteSolutions`.
- ◆ In the Callable library, use the routine `CPXsolwritessolnpoolall`.
- ◆ In the Interactive Optimizer, use this command: `write filename .sol all`

Similarly, you can use a solution from the solution pool to change the fixed problem of your MIP model. Only these two types are supported for this change:

- ◆ `CPXPROB_FIXEDMILP`
- ◆ `CPXPROB_FIXEDMIQP`
- ◆ In Concert Technology
 - In the C++ API, use `IloCplex::solveFixed`.
 - In the Java API, use `IloCplex.solveFixed`.
 - In the .NET API, use `Cplex.SolveFixed`.
- ◆ In the Callable Library (C API), use the routine `CPXchgprobtypesolnpool`.
- ◆ In the Interactive Optimizer, the following command changes the fixed problem to that of the solution at index *i* in the pool: `change problem fixed i`

The parameter *write level for MST, SOL files* (`WriteLevel`, `CPX_PARAM_WRITELEVEL`), documented in the *ILOG CPLEX Parameter Reference Manual*, enables you to specify various levels of information, such as values for only discrete variables, values for all variables, and so forth, for CPLEX to record about a solution when it writes the solution to a formatted file.

Deleting solutions from the solution pool

When the *advanced start switch* (`AdvInd`, `CPX_PARAM_ADVIND`) is set to 0 (zero) and a new optimization is started, either by MIP optimization or by `populate`, all solutions in the solution pool are deleted automatically.

You can also delete solutions yourself from the pool.

- ◆ In Concert Technology, use the methods:
 - `delSolnPoolSolns` in the C++ API.
 - `delSolnPoolSolns(int, int)` in the Java API.
 - `Cplex.DelSolnPoolSolns` in the .NET API.
- ◆ In the Callable Library (C API) use the routine `CPXdelSolnPoolSolns` to delete solutions from the solution pool.
- ◆ In the Interactive Optimizer, use this command, where *i* specifies the index of the solution to be deleted: `change delete solutions i`

The incumbent and the solution pool

A copy of the incumbent solution (that is, the best integer solution found relative to the objective function) is always added to the pool, as long as the pool capacity is at least one, regardless of its evaluation with respect to any filters and regardless of the replacement criterion governing the solution pool. This copy of the incumbent solution will be the first member of the solution pool, that is, the solution with index 0 (zero). The incumbent is accessible through queries that use the symbolic value identifying the incumbent.

- ◆ In the C++ API, use the value `IloCplex::IncumbentId` as an argument to such methods as `IloCplex::getValues`, `getSlack`, `getSlacks`, `getQuality`, `getObjValue`.
- ◆ In the Java API, use the value `IloCplex.IncumbentId` as an argument to such methods as `IloCplex.getValues`, `getSlack`, `getSlacks`, `getQuality`, `getObjValue`.
- ◆ In the .NET API, use the value `Cplex.IncumbentId` as an argument to such methods as `Cplex.GetValues`, `GetSlack`, `GetSlacks`, `GetQuality`, `GetObjValue`.
- ◆ In the Callable Library (C API), use the symbolic value `CPX_INSUMBENT_ID` as an argument to such routines as, `CPXgetsolnpoolx`, `CPXgetsolnpoolobjval`, `CPXgetsolnpoolslack`, `CPXgetsolnpoolqconstrslack`, `CPXgetsolnpooldblquality`, `CPXgetsolnpoolintquality`, `CPXgetsolnpoolsolnname`, `CPXsolwritesolnpool`, `CPXchgprobtypesolnpool`.
- ◆ In the Interactive Optimizer use the command `display solution member 0` or `display solution member incumbent`. That is, you can display the incumbent solution by index number or by name.

Parameters of the solution pool

Describes parameters to control the solution pool.

In this section

Which parameters control the solution pool?

Lists parameters to control the solution pool.

Example: quality control through the solution pool gap parameter

Illustrates quality control through the solution pool gap parameter.

Example: few or many solutions through intensity parameter

Illustrates effect of the intensity parameter on the solution pool.

Example: diverse solutions through replacement parameter

Illustrates effect on diversity of the replacement parameter of the solution pool.

Which parameters control the solution pool?

ILOG CPLEX provides parameters to control the solution pool. The table titled *Parameters for the solution pool* summarizes these parameters. *ILOG CPLEX Parameter Reference Manual* documents each of these parameters in greater detail.

Parameters for the solution pool

Purpose	Concert parameter	Callable Library parameter	Interactive Optimizer	Parameter Reference
Intensity	SolnPoolIntensity	CPX_PARAM_SOLNPOOLINTENSITY	mip pool intensity	<i>solution pool intensity</i>
Limit on populate (number of solutions generated)	PopulateLim	CPX_PARAM_POPULATELIM	mip limits populate	<i>limit on number of solutions generated for solution pool</i>
Limit on capacity (number of solutions stored)	SolnPoolCapacity	CPX_PARAM_SOLNPOOLCAPACITY	mip pool capacity	<i>limit on number of solutions kept in solution pool</i>
Replacement strategy	SolnPoolReplace	CPX_PARAM_SOLNPOOLREPLACE	mip pool replace	<i>solution pool replacement strategy</i>
Relative objective gap	SolnPoolGap	CPX_PARAM_SOLNPOOLGAP	mip pool relgap	<i>relative gap for solution pool</i>
Absolute objective gap	SolnPoolAGap	CPX_PARAM_SOLNPOOLAGAP	mip pool absgap	<i>absolute gap for solution pool</i>

The *MIP node limit* parameter (NodeLim, CPX_PARAM_NODELIM) and *optimizer time limit* parameter (TiLim, CPX_PARAM_TILIM) also have an effect on the solution pool, just as they influence MIP optimization generally. For more detail about these parameters, see their entries in the *ILOG CPLEX Parameter Reference Manual*.

Example: quality control through the solution pool gap parameter

In many cases, solutions are interesting only if their objective value is close to the optimal objective value of the model. In that context, you can control the quality of solutions generated and saved in the solution pool with the *absolute gap for solution pool* parameter (SolnPoolAGap, CPX_PARAM_SOLNPOOLAGAP) and the *relative gap for solution pool* parameter (SolnPoolGap, CPX_PARAM_SOLNPOOLGAP).

To demonstrate this idea, consider again the example cited in *Example: simple facility location problem*. In order to obtain solutions that are less than 10% worse than the optimal objective value, specify the solution pool relative gap parameter in the Interactive Optimizer like this:

```
read location.lp
set mip pool relgap 0.1
populate
```

Then display the objective value of each solution in the Interactive Optimizer with this command:

```
display solution list *
```

Afterwards, you see that all solutions in the pool are of a cost less than or equal to 548; that is, within 10% of the optimal objective value of 499.

Example: few or many solutions through intensity parameter

Use the *solution pool intensity* parameter (`SolnPoolIntensity`, `CPX_PARAM_SOLNPOOLINTENSITY`) to balance the number of solutions generated and the amount of time or memory consumed. Lower intensity values generate fewer solutions, whereas higher intensity values generate more solutions.

If you need many solutions, but do not want to impair performance too greatly, the value 2 (moderate) is a good choice for most models.

For example, the following session in the Interactive Optimizer reads a model in LP format of the *Example: simple facility location problem*. The session then effectively removes the stopping criterion of the populate limit parameter by setting it very high.

```
read location.lp
set mip limits populate 10000
set mip pool intensity 2
set mip pool relgap 0.1
populate
```

You can see from the log that setting the pool intensity to 2 yields results faster than when populate is called after MIP optimization at the default value of solution pool intensity.

If you set the solution pool intensity parameter to 3 (aggressive) or 4 (very aggressive), then an even larger number of solutions will be produced.

At solution pool intensity 2, a large number of solutions are produced (in this case, 196 solutions, though the precise number of solutions may vary on your platform).

If you set solution pool intensity at 3 instead, populate will generate a greater number of solutions (in this case, 208 solutions).

Likewise, if you set solution pool intensity at 4 instead, a very great number of solutions will be produced and stored in the solution pool, as this setting exhaustively enumerates solutions.

In this small example, the settings 3 and 4 happen to produce the same number of solutions (208), but in general it will not be the case that the two settings have the same effect.

Example: diverse solutions through replacement parameter

It is often impractical to manage a very large number of solutions, and in those situations, a smaller set of solutions with different characteristics proves more useful. You can achieve this aim in two steps:

1. Set the solution pool capacity parameter (*limit on number of solutions kept in solution pool* `SolnPoolCapacity`, `CPX_PARAM_SOLNPOOLCAPACITY`) to a manageable number, rather than its default value, which is quite large.
2. Set the *solution pool replacement strategy* (`SolnPoolReplace`, `CPX_PARAM_SOLNPOOLREPLACE`) to 2: replace least diverse solutions.

These settings make sure that pool capacity will not increase as solutions are added. Instead, solutions will be replaced in the pool according to their diversity if the number of solutions generated exceeds the limited capacity of the pool.

As an example of this idea of using the replacement strategy parameter to control diversity of solutions in the solution pool, consider the following session in the Interactive Optimizer, again reading the model in *Example: simple facility location problem*, setting parameters, and calling `populate`.

```
read location.lp
set mip pool intensity 2
set mip pool relgap 0.1
set mip pool replace 2
set mip pool capacity 10
set mip limits populate 10000
set time 1
populate
```

Logically, the pool contains only ten solutions now (its capacity) even though more solutions have been generated. The number of solutions that have been generated but are not retained in the pool is reported in the log by the number of solutions replaced.

If you apply a time limit of 10 seconds instead of 1 (one), many more solutions will be generated. That greater number of solutions also leads to greater diversity among the solutions retained in the pool.

Filtering the solution pool

Documents filters of the solution pool. Filters offer a means of controlling properties of the solutions you generate and store.

In this section

What are filters of the solution pool?

Defines filtering of the solution pool.

Diversity filters

Describes diversity filters of the solution pool.

Range filters

Describes range filters of the solution pool.

Filter files

Describes filter files for the solution pool.

Example: controlling properties of solutions with filters

Illustrates filters in use in the solution pool.

Incumbent callback as a filter

Describes the incumbent callback as a filter for the solution pool.

What are filters of the solution pool?

Filtering allows you to control properties of the solutions generated and stored in the solution pool. ILOG CPLEX provides two predefined ways to filter solutions.

- ◆ If you want to filter solutions based on their difference as compared to a reference solution, use a diversity filter, as explained in *Diversity filters*.
- ◆ If you want to filter solutions based on their validity in an additional linear constraint, use a range filter, as explained in *Range filters*.

Those two ways are practical for most purposes. However, if you require finer control of which solutions to keep and which to eliminate, use an incumbent callback, as explained in *Incumbent callback as a filter*.

Adding or deleting filters does not affect the solutions already in the pool; new filters are applied only at the next call of MIP optimization or the populate procedure.

Filters are assigned an index number when they are added to the solution pool, and they may be accessed through this index number. Diversity filters and range filters share the same sequence of indices.

To count the number of filters associated with the solution pool, use one of these methods, routines, or commands:

- ◆ In Concert Technology
 - `getNfilters` in the C++ API;
 - `IloCplex.getNFilters` in the Java API;
 - `Cplex.GetNFilters` in the .NET API.
- ◆ `CPXgetsolnpoolnumfilters` in the Callable Library (C API);

To access a filter, use one of these methods, routines, or commands:

- ◆ Concert Technology offers a method to access each characteristic of a filter; for example, `getFilterType`, `getFilterIndex`, `getFilterVars`, and so forth.
- ◆ `CPXgetsolnpoolrngfilter` or `CPXgetsolnpooldivfilter` in the C API;

To delete a filter, use one of these methods, routines, or commands:

- ◆ In Concert Technology
 - In the C++ API, use the method `IloCplex::delFilter`.
 - In the Java API, use the method `IloCplex.delFilter`.
 - In the .NET API, use the method `Cplex.DelFilter`.

- ◆ In the Callable Library (C API), use `CPXdelsolnpoolfilters`.
- ◆ In the Interactive Optimizer, use the command `change delete filter`.

Diversity filters

A *diversity filter* allows you generate solutions that are similar to (or different from) a set of *reference values* that you specify for a set of binary variables. In particular, you can use a diversity filter to generate more solutions that are similar to an existing solution or to an existing partial solution. Several diversity filters can be used simultaneously, for example, to generate solutions that share the characteristics of several different solutions.

To create a diversity filter, use one of these methods, routines, or commands.

- ◆ In Concert Technology
 - In the C++ API, use the method `IloCplex::addDiversityFilter` to add a diversity filter to the invoking `IloCplex` object.
 - In the Java API, use the method `IloCplex.addDiversityFilter` to add a diversity filter to the invoking `IloCplex` object.
 - In the .NET API, use the method `Cplex.AddDiversityFilter` to add a diversity filter to the invoking `Cplex` object.
- ◆ In the Callable Library (C API), use the routine `CPXaddsolnpooldivfilter`, passing the reference values, variables of interest, and weights as arguments; or, use your favorite text editor to create a file formatted according to the specifications in *FLT file format: filter files for the solution pool*, and then install the contents of that file with the routine `CPXreadcopysolnpoolfilters`.
- ◆ In the Interactive Optimizer, use your favorite text editor to create a file formatted according to the specifications in *FLT file format: filter files for the solution pool*; then install the contents of that formatted file with the command: `read filename flt`

For greater detail about the characteristics of diversity filters and reference sets, see the documentation of those methods and routine in the *Reference Manuals* of the APIs. For an example of a filter in use, see *Example: controlling properties of solutions with filters*.

Range filters

A *range filter* allows you to generate solutions that obey a new constraint, specified as a linear expression within a range.

The difference between adding a range filter and adding a linear constraint directly to the model is that you can add range filters without losing information computed in previous invocations of MIP optimization or populate and stored in the search space. In contrast, if you change the model directly by adding constraints, the next call of optimize or populate will discard previous information and restart from scratch on the changed model.

Range filters can be used to express diversity constraints that are more complex than the standard form implemented by diversity filters. In particular, range filters also apply to general integer variables, semi-integer variables, continuous variables, and semi-continuous variables, not just to binary variables.

To create a range filter, use one of the following methods, routines, or commands.

- ◆ In Concert Technology
 - In the C++ API, use the method `IloCplex::addRangeFilter` to add the range filter to the invoking instance of `IloCplex`.
 - In the Java API, use the method `IloCplex.addRangeFilter` to add the range filter to the invoking instance of `IloCplex`.
 - In the .NET API, use the method `Cplex.AddRangeFilter` to add the range filter to the invoking instance of `Cplex`.
- ◆ In the Callable Library (C API), use the routine `CPXaddsolnpoolrngfilter`, passing a linear constraint and range as arguments; or, use your favorite text editor to create a file formatted according to the specifications in *FLT file format: filter files for the solution pool*, and then install the contents of that file with the routine `CPXreadcopysolnpoolfilters`.
- ◆ In the Interactive Optimizer, use your favorite text editor to create a file formatted according to the specifications in *FLT file format: filter files for the solution pool*; then install the contents of that formatted file with the command: `read filename.flr`

For more detail about the characteristics of range filters, see the documentation of those methods and routine in the *Reference Manuals* of the APIs. For an example of a filter in use, see *Example: controlling properties of solutions with filters*.

Filter files

You can store filters in a file, known as a *filter file*, distinguished by the file extension `.flt`. The same filter file can contain several filters, including both diversity filters and range filters. For documentation of the format of a filter file, see *FLT file format: filter files for the solution pool* in the *ILOG CPLEX File Format Reference Manual*.

To create filters, use your favorite text editor to create a file formatted according to the specifications in *FLT file format: filter files for the solution pool*.

To install filters declared in a filter file, use one of these methods, routines, or commands:

◆ In Concert Technology, use the methods:

- `IloCplex::readFilters`
- `IloCplex.readFilters`
- `Cplex.ReadFilters`

◆ In the Callable Library (C API), use the routine `CPXreadcopysolnpoolfilters` to add diversity or range filters.

◆ In the Interactive Optimizer, use the `read` command to import a filter file.

To write existing filters to a formatted file (for example, for re-use later), use these methods, routines, or commands:

◆ In Concert Technology, use the methods:

- `writeFilters` in the C++ API;
- `writeFilters(String)` in the Java API;
- `Cplex.WriteFilters` in the .NET API.

◆ In the Callable Library (C API), use the routine `CPXfltwrite`.

◆ In the Interactive Optimizer, use the `write` command with the file type option `flt` to create a filter file of the filters currently associated with the solution pool. For example, the following command creates a file named `filename.flt` containing the filters associated with the solution pool: `write filename flt`

Example: controlling properties of solutions with filters

The model in *Example: simple facility location problem* has two categories of variables. The x variables specifying the facilities to open are of a higher decision level than the y variables deciding how the goods are shipped from facilities to regions. Suppose, for example, that you want to populate the solution pool with solutions that differ by which facilities are opened, without specifying any specific criteria for the shipping decisions. The replacement strategy (shown in *Example: diverse solutions through replacement parameter*) does not allow you to specify a customized diversity measure that takes into account only a subset of the variables. However, this diversity measure expressed only over the x variables can be enforced through a diversity filter.

Suppose further that facilities 1 and 2 are open. Let a solution keeping those two facilities open be the reference; that is, the reference value for x_1 is 1 (one), for x_2 is 1 (one), for x_3 is 0 (zero), for x_4 is 0 (zero). Then use a diversity filter to stipulate that any solution added to the solution pool must differ from the reference by decisions to open or close at least two other facilities. The following filter file enforces this diversity by giving each x variable a weight of 1.0 and specifying a minimum diversity of 2 and unlimited maximum diversity (that is, infinity). In other words, this diversity filter makes sure that solutions satisfy the following constraint:

```
2 <= 1.0 * |x1 - 1| + 1.0 * |x2 - 1| + 1.0 * |x3 - 0| + 1.0 * |x4 - 0| <=
infinity
```

The y variables are not specified in the filter; hence, they are not taken into account in the diversification.

```
NAME location
DIVFILTER f1 2 inf
x1 1.0 1
x2 1.0 1
x3 1.0 0
x4 1.0 0
ENDATA
```

Range filters also enforce additional constraints. Suppose, for example, that you want to limit transportation costs to less than fixed costs. The following range filter enforces this restriction by expressing the linear constraint:

```
-infinity <= 1.0 * transport - 1.0 * fixed <=0
```

```
NAME location
RNGFILTER f2 -inf 0
transport 1.0
```

```
fixed -1.0  
ENDATA
```

Incumbent callback as a filter

If you need to enforce more complex constraints on solutions (if you need to enforce nonlinear constraints, for example), you can use the incumbent callback in Concert Technology or the Callable Library. During the populate procedure, the incumbent callback is called each time a new solution is found, even if the new solution does not improve the objective value of the incumbent. The incumbent callback allows your application to accept or reject the new solution based on your own criteria.

Bear in mind that the incumbent callback disables dynamic search. At default parameter settings, the incumbent callback, as a control callback, also disables deterministic parallel MIP optimization (if your application is licensed for parallel MIP optimization) though you can override this default behavior by setting the *parallel mode switch* (`ParallelMode`, `CPX_PARAM_PARALLELMODE`) yourself. For more about dynamic search, see *Branch & cut or dynamic search?*. For more about parallel MIP optimization, see *Parallel MIP optimizer*.

To create an incumbent callback, use the following methods or routine:

- ◆ In Concert Technology:
 - In the C++ API, derive a subclass of `IloCplex::IncumbentCallbackI`, the implementation class documented in the reference manual. Use your user-defined subclass according to the instructions in the parent class `IloCplex::CallbackI`.
 - In the Java API, implement a subclass of `IloCplex.IncumbentCallback`. Use it as documented in the parent class `IloCplex.Callback`.
 - In the .NET API, implement a subclass of `Cplex.IncumbentCallback`. Use it as documented in the parent class `Cplex.Callback`.
- ◆ In the Callable Library (C API), use the routine `CPXsetincumbentcallbackfunc`.

Callbacks typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. For more information, please read *Using optimization callbacks* and *Advanced MIP control interface*.

Using special ordered sets (SOS)

Describes special ordered sets (SOSs) in a model as a way to specify integrality conditions.

In this section

What is a special ordered set (SOS)?

Defines a special ordered set, explains its purpose, and describes its effect.

Example: SOS Type 1 for sizing a warehouse

Illustrates special ordered sets in a warehouse example.

Declaring SOS members

Describes routines and methods for declaring a special ordered set.

Example: using SOS and priority

Illustrates how to integrate priority orders with a special ordered set.

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*, used the coefficients of the warehouse capacity constraint to assign weights.

Example: using SOS and priority

Illustrates how to integrate priority orders with a special ordered set.

In this section

ilomipex3.cpp

Illustrates priority orders and a special ordered set in the C++ API.

mipex3.c

Illustrates priority orders and a special ordered set in the C API.

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

Demonstrates semi-continuous variables in Concert Technology in an example of managing production in a power plant.

In this section

What are semi-continuous variables?

Defines semi-continuous variables.

Describing the problem

Describes a power plant to illustrate a model with semi-continuous variables.

Representing the problem

Identifies salient features of the model with semi-continuous variables.

Building a model

Describes the application to solve a model with semi-continuous variables.

Solving the problem

Describes activity in the application.

Ending the application

Describes memory management in the application.

Complete program

Identifies location of the sample application in the C++ API and other APIs as well.

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 result in that bound being treated as 0 (zero).

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 specify 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_SEMCONT` 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* 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.

```
if (cplex.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

Demonstrates the use of piecewise linear functions to solve a transportation problem.

In this section

What is a piecewise linear function?

Defines a piecewise linear function.

Syntax of piecewise linear functions

Describes the syntax to represent a piecewise linear function.

Discontinuous piecewise linear functions

Defines a discontinuous piecewise linear function.

Isolated points in piecewise linear functions

Defines an isolated point in a piecewise linear function.

Using `IloPiecewiseLinear` in expressions

Describes a piecewise linear function as an expression.

Describing the problem

Demonstrates a problem for which piecewise linear functions are suitable.

Developing a model

Describes an application using piecewise linear functions.

Solving the problem

Describes activity in the application.

Displaying a solution

Describes display of the solution from the application.

Ending the application

Describes memory management in the application.

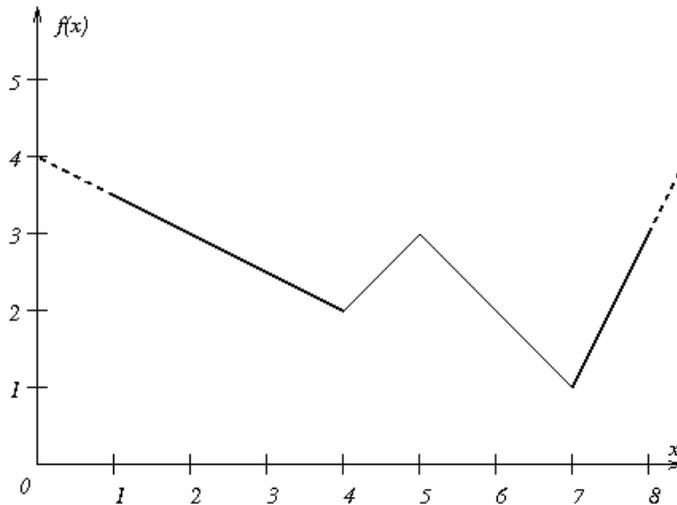
Complete program: transport.cpp

Tells where the sample application in C++ is located and where to find variations in other APIs.

What is a piecewise linear function?

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 topic, 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.

From a geometric point of view, *A piecewise linear function with breakpoints* 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 *A piecewise linear function with breakpoints* has three breakpoints.



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 *A piecewise linear function with breakpoints* are specified in this way, where the first argument, `env`, specifies the environment, the second argument specifies the number of breakpoints under consideration, and the remaining arguments specify the x -coordinate of the breakpoints:

```
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, where the first argument again specifies the environment, the second argument specifies the number of slopes given, and the remaining arguments specify the slope of the segments:

```
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 *A piecewise linear function with breakpoints* looks like this:

```
IloPiecewiseLinear(x, -0.5, IloNumArray(env, 3, 4., 5., 7.),  
                    IloNumArray(env, 3, 2., 3., 1.), 2);
```

Note: It may help you understand the signatures of these functions to recall the familiar Cartesian representation of a line or segment in two dimensions, x and y :

$$y = ax + b$$

where a represents the slope of the line or segment and b represents the height at which the line theoretically crosses the y -axis at the point $(0, b)$.

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 *A piecewise linear function with breakpoints*.

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. *A discontinuous piecewise linear function with steps* shows a discontinuous piecewise linear function with two steps.

Syntactically, a step is represented in this way:

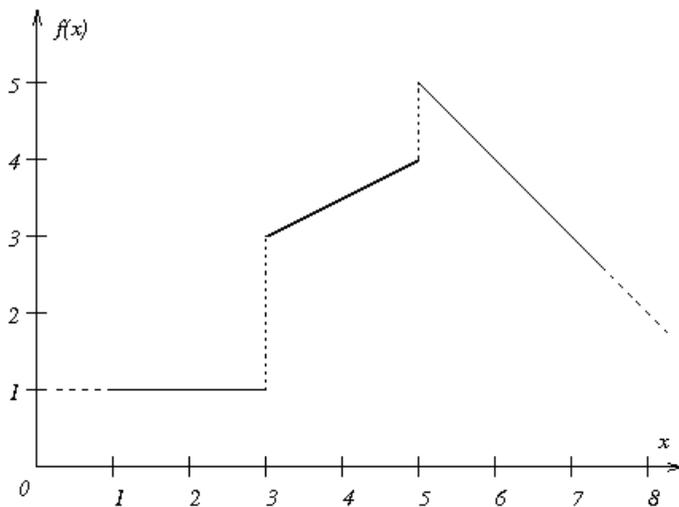
- ◆ The x -coordinate of the breakpoint where the step occurs is repeated in the array of the breakpoint.
- ◆ The value of the first point of a step in the array of slopes is the height of the step.
- ◆ The value of the second point of the step in the array of slopes is the slope of the function after the step.

By convention, a breakpoint belongs in both segments associated with the step. For example, in *A discontinuous piecewise linear function with steps*, 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*, 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 *A discontinuous piecewise linear function with steps* 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);
```



A discontinuous piecewise linear function with steps

Note: It may help to understand the signature of the function in this example to recall that in Cartesian coordinates, the slope of a horizontal line (that is, a line or segment parallel to the x-axis) is 0 (zero), and the slope of a vertical line (that is, a line or segment parallel to the y-axis) is undefined in the conventional representation:

$$y = ax + b$$

In the signature of the function, the undefined slope of the vertical segment at the point of discontinuity is represented as the height of the step.

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 *A discontinuous piecewise linear function with steps* 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` in expressions

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

Demonstrates a problem for which piecewise linear functions are suitable.

In this section

Problem statement

Describes a model using piecewise linear functions.

Variable shipping costs

Distinguishes convex from concave piecewise linear functions.

Model with varying costs

Describes syntax to represent varying costs in the model.

Problem statement

Assume that a company must ship cars from factories to showrooms. Each factory can supply a fixed number of cars, and each showroom needs a fixed number of cars. There is a cost for shipping a car from a given factory to a given showroom. The objective is to minimize the total shipping cost while satisfying the demands and respecting supply.

In concrete terms, assume there are three factories and four showrooms. Here is the quantity that each factory can supply:

```
supply0 = 1000
supply1 = 850
supply2 = 1250
```

Each showroom has a fixed demand:

```
demand0 = 900
demand1 = 1200
demand2 = 600
demand3 = 400
```

Let `nbSupply` be the number of factories and `nbDemand` be the number of showrooms. Let x_{ij} be the number of cars shipped from factory i to showroom j . The model is composed of `nbDemand` + `nbSupply` constraints that force all demands to be satisfied and all supplies to be shipped. Thus far, a model for our problem looks like this:

Minimize

$$\sum_{i=0}^{nbDemand-1} \sum_{j=0}^{nbSupply-1} cost_{ij} \cdot x_{ij}$$

subject to

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

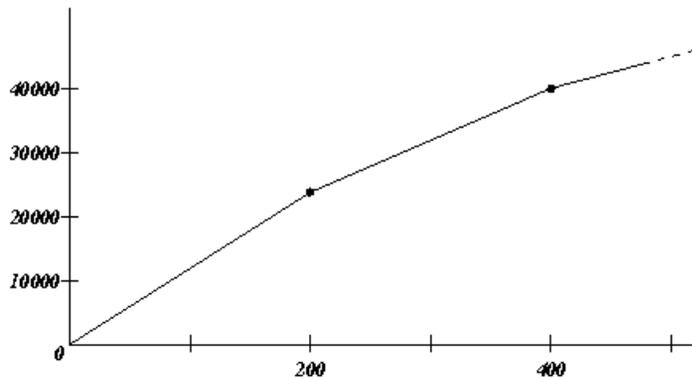
$$\sum_{i=0}^{nbDemand-1} x_{ij} = demand_j \quad j = 0, \dots, nbSupply-1$$

Variable shipping costs

Now consider the costs of shipping from a given factory to a given showroom. Assume that for every pair (factory, showroom), there are different rates, varying according to the quantity shipped. To illustrate the difference between convex and concave piecewise linear functions, in fact, this example assumes that there are two different tables of rates for shipping cars from factories to showrooms. The first table of rates looks like this:

- ◆ a rate of 120 per car for quantities between 0 and 200;
- ◆ a rate of 80 per car for quantities between 200 and 400;
- ◆ a rate of 50 per car for quantities higher than 400.

These costs that vary according to quantity define the piecewise linear function represented in *A concave piecewise linear cost function*. As you see, the slopes of the segments of that function are decreasing, so that function is concave.

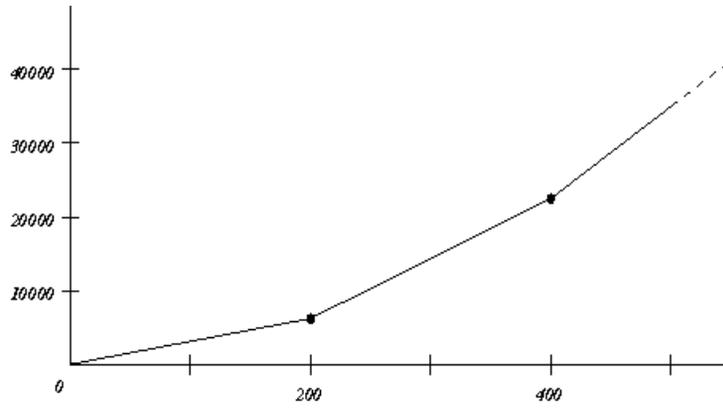


A concave piecewise linear cost function

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 *A convex piecewise linear cost function*. The slopes of the segments in this second piecewise linear function are increasing, so this function is convex.



A convex piecewise linear cost function

Model with varying costs

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

Minimize

$$\sum_{i=0}^{nbDemand-1} \sum_{j=0}^{nbSupply-1} y_{ij}$$

subject to

$$y_{ij} = f(x_{ij}) \text{ for } i = 0, \dots, nbDemand-1 \text{ and } j = 0, \dots, nbSupply-1$$

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

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

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

Developing a model

Describes an application using piecewise linear functions.

In this section

Creating the environment and model

Describes creation of the environment and model in the application.

Representing the data

Describes populating the model with data.

Adding constraints

Describes adding constraints in the application.

Checking convexity and concavity

Describes data checking in the application to detect convexity and concavity.

Adding an objective

Describes adding an objective function to the application.

Creating the environment and 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

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

Describes logical constraints in ILOG CPLEX with Concert Technology.

In this section

What are logical constraints?

Defines logical constraints.

What can be extracted from a model with logical constraints?

Documents the logical constraints available in each API.

Which nonlinear expressions can be extracted?

Describes certain nonlinear expressions as logical constraints.

Logical constraints for counting

Describes cardinality (counting) as a logical constraint.

Logical constraints as binary variables

Describes logical constraints as terms in expressions.

How are logical constraints extracted?

Describes extraction of logical constraints as indicators.

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?*.

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 *Indicator constraints in optimization* in this manual.

What can be extracted from a model with logical constraints?

Documents the logical constraints available in each API.

In this section

Overview

Introduces logical constraints in the context of extraction.

Logical constraints in the C++ API

Describes logical constraints in the C++ API.

Logical constraints in the Java API

Describes logical constraints in the Java API.

Logical constraints in the .NET API

Describes logical constraints in the .NET API.

Overview

Concert Technology offers classes for you to design a model of your problem, of course. You can then invoke an algorithm to *extract* information from your model to solve the problem. In this context, an *algorithm* is an instance of a class such as `IloCplex`, documented in the *ILOG CPLEX Reference Manuals* of the C++ and Java APIs, or of the class `CPLEX`, documented in the *ILOG CPLEX Reference Manual* of the .NET API. For more about this idea of extraction generally, see topics in *Languages and APIs*, or see the concept of Extraction in the *ILOG CPLEX Reference Manual* of the C++ API.

When your model contains certain logical operators and conditional statements, Concert Technology extracts them as logical constraints appropriately. Much the same logical constraints are available in these APIs of ILOG CPLEX.

- ◆ *Logical constraints in the C++ API*
- ◆ *Logical constraints in the Java API*
- ◆ *Logical constraints in the .NET API*

For similar facilities in the **Callable Library**, see *Indicator constraints in optimization*.

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*, summarized in *Concert Technology modeling objects in C++*. 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 $x^2 + y^2$. 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 *Indicator constraints in optimization*.

Indicator constraints in optimization

Introduces indicator constraints and emphasizes their advantages over Big M formulations..

In this section

What is an indicator constraint?

Defines indicator constraints.

Example: fixnet.c

Illustrates indicator constraints in an application of the C API; contrasts indicator constraints with Big M formulations.

Indicator constraints in the Interactive Optimizer

Contrasts indicator constraints with a Big M formulation in the Interactive Optimizer.

What are indicator variables?

Defines an indicator variable and directs you to an example.

Restrictions on indicator constraints

Describes limitations on indicator constraints.

Best practices with indicator constraints

Suggests best practices with respect to indicator constraints.

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*.

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 itself

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. In that respect, the use of indicator constraints instead of a Big M formulation offers a more numerically stable model, closer to the mathematical programming issues of the problem, and thus more likely to produce useful solutions of the problem.

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) is 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

Demonstrates logical constraints in a sample application.

In this section

Introducing the example

Introduces an example from food manufacturing to illustrate logical constraints.

Describing the problem

Describes the problem verbally.

Representing the data

Raises questions pertinent to a sound representation of the data of the problem.

Developing the model

Describes creation of the model for the problem.

Formulating logical constraints

Contrasts conventional formulation of constraints with more efficient logical constraints.

Solving the problem

Describes activity in the application, display of the solution, and memory management.

Introducing the example

Logical constraints in optimization introduced features of ILOG CPLEX that transform parts of your problem automatically for you. This topic shows you some of those features in use in a C++ application. The example 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.

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 about the problem?
- ◆ What are the unknowns of the problem?
- ◆ What are the constraints of the problem?
- ◆ What is the objective of the problem?

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*, there are various 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 *Representing the data: What are the constraints?*), 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));
```

Formulating logical constraints

You have already seen how to represent the logical constraints of this problem in *Representing the data: What are the constraints?* However, they deserve a second glance because they illustrate an important point about logical constraints and their 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 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;
```

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

Solves a scheduling problem by applying logical constraints, piecewise linear functions, and aggressive MIP emphasis.

In this section

Describing the problem

Describes the problem to solve in the application.

Understanding the data file

Describes the data of the problem.

Reading the data

Shows how to read data from a file to fill matrices in the model.

Creating variables

Identifies unknowns of the problem.

Stating precedence constraints

Shows syntax of precedence constraints for this model.

Stating resource constraints

Shows syntax of resource constraints for this model.

Representing the piecewise linear cost function

Shows the earliness-tardiness cost function as a piecewise linear function.

Transforming the problem

Explains what ILOG CPLEX extracts.

Solving the problem

Highlights MIP emphasis parameter and tells where to find the application.

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:

```
s1 >= s2 + d2 or s2 >= s1 + d1
```

where s is the starting date of an activity and d is its duration.

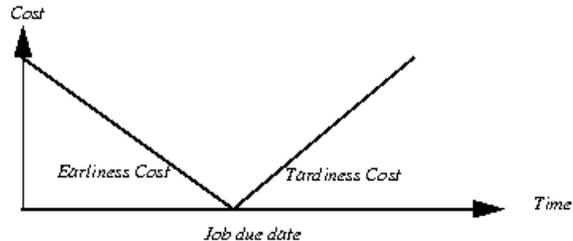
If n activities need to be processed on the same resource then about $(n*n) / 2$ disjunctions need to be stated and added to the model, like this:

```
for(i = 0; i < nbResource; i++) {
  IloInt end = nbJob - 1;
  for(j = 0; j < end; j++){
    IloInt a = activityOnAResource[i][j];
    for(IloInt k = j + 1; k < nbJob; k++){
      IloInt b = activityOnAResource[i][k];
      model.add(s[j][a] >= s[k][b] + duration[k][b]
        ||
        s[k][b] >= s[j][a] + duration[j][a]);
    }
  }
}
```

Representing the piecewise linear cost function

The earliness-tardiness cost function is the sum of piecewise linear functions having two segments, as you see in *Earliness and tardiness as piecewise linear cost function*. 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));
```



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.

Solution polishing is also useful in this problem. Activate solution polishing by specifying a positive value for the parameter that controls the amount of time (in seconds) spent on polishing. For detail about that parameter, see *time before starting to polish a feasible solution* in the *ILOG CPLEX Parameters Reference Manual*. For more detail about invoking solution polishing, see *Solution polishing* in this manual.

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

Uses an example of cutting stock to demonstrate the technique of column generation in Concert Technology.

In this section

What is column generation?

Defines column generation.

Column-wise models in Concert Technology

Describes features of Concert Technology to support column generation.

Describing the problem

Describes a problem to illustrate column generation.

Representing the data

Describes environment, model, and data in the application.

Developing the model: building and modifying

Describes column generation in this application with generalizations about other applications.

Changing the objective function

Describes modification of the objective function in the application.

Solving the problem: using more than one algorithm

Uses two instances of the algorithm to solve two models (master and column generator).

Ending the program

Ends the application with memory management in the C++ API.

Complete program

Tells where to find the sample application in the C++ API and other variations in other APIs.

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 discover 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 decide 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 the `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 `IloCP`, of `IloCplex` itself, or of other subclasses of `IloAlgorithm`, for example) lend themselves to other parts of the problem by specifying which variables to consider next (and thus which columns to generate).

In the *ILOG CPLEX 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.



Two different patterns from a roll of stock

Even with that simplifying assumption, the fact that there can be so many different patterns makes a naive model of this problem (where a user declares one variable for every possible pattern) impractical. Such a model introduces too many symmetries. Fortunately, for any given customer order, a limited number of patterns will suffice, so many of the possible patterns can be disregarded, and the application can focus on finding the relevant ones.

Here is a conventional statement of a cutting stock problem in terms of the unknown X_j , the number of times that pattern j will be used, and A_{ij} , the number of items i of each pattern j needed to satisfy demand d_i :

Minimize:

$$\sum_j X_j$$

subject to:

$$\sum_{i \neq j} A_{ij} X_j$$

with:

$$X_j \geq 0$$

Solving this model with all columns present from the beginning is practically impossible. In fact, even with only 10 types of items with a size roughly 1/10 of the width of the roll, there would exist roughly 10^{10} kinds of patterns, and hence that many decision variables. Such a formulation might not even fit in memory on a reasonably large computer. Moreover, most of those patterns would obviously not be interesting in a solution. These considerations make column generation an interesting approach for this problem.

To solve a cutting stock problem by column generation, start with a subproblem. Choose one pattern, lay it out on the stock, and cut as many items as possible, subject to the constraints of demand for that item and the width of the stock. This procedure will surely work in that it produces some answer (a feasible solution) to the problem, but it will not necessarily produce a satisfactory answer in this way 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:

Minimize:

$$1 - \sum_i \pi_i A_i$$

subject to:

$$\sum_i A_i \leq W$$

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

Describes column generation in this application with generalizations about other applications.

In this section

The master model and column generator in this application.

Outlines the design of master model and column generator with emphasis on modifications during model building.

Adding extractable objects: both ways

Describes alternative ways to add columns to the model.

Adding columns to a model

Shows how to add generated columns to a master model.

Changing the type of a variable

Describes modifying the type of variables in the master model to support integrality constraints.

Cut optimization model

Describes the initial model in the application.

Pattern generator model

Describes the column generator in the application.

The master model and column generator in this application.

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 `IloCP`) about the modification. (For more about that idea, see the concept of *Notification* in the *ILOG CPLEX 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

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 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, nWdth, 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 of `IloEnv::end` to de-allocate the models and algorithms 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/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*.

Infeasibility and unboundedness

Documents tools to help you analyze the source of the infeasibility in a model: the preprocessing reduction parameter for distinguishing infeasibility from unboundedness, the conflict refiner for detecting minimal sets of mutually contradictory bounds and constraints, and FeasOpt for repairing infeasibilities.

In this section

Preprocessing and feasibility

Introduces the detection of infeasibilities during preprocessing.

Managing unboundedness

Discusses tactics to diagnose the cause of an unbounded outcome in the optimization of a model and suggests ways to avoid an unbounded outcome.

Diagnosing infeasibility by refining conflicts

Describes 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.

Repairing infeasibilities with FeasOpt

Documents FeasOpt, a feature for repairing infeasibility in a model.

Preprocessing and feasibility

Introduces the detection of infeasibilities during preprocessing.

In this section

Issues of infeasibility and unboundedness

Outlines issues of infeasibility and unboundedness.

Early reports of infeasibility based on preprocessing reductions

Explains how to interpret early reports of infeasibility and how to use a parameter in diagnosis.

Issues of infeasibility and unboundedness

The topics discussed in *Continuous optimization* and *Discrete optimization* often contained the implicit assumption that a bounded feasible solution to your model actually exists. Such may not always be the case.

The following topics discuss 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?*.)

Infeasibility and unboundedness are closely related topics in optimization theory. Consequently, certain of the concepts for one will have direct relation to the other. As you know, ILOG CPLEX can provide you solution information about the models that it optimizes. For infeasible outcomes, it reports values that you can analyze to detect 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 for you to try to narrow down the investigation with help from ILOG CPLEX, for example, through tools such as the conflict refiner, documented in *Diagnosing infeasibility by refining conflicts*. In certain situations, ILOG CPLEX even provides some degree of automatic repair, for example, with tools such as FeasOpt, documented in *Repairing infeasibilities with FeasOpt*.

The following topic explains how to interpret reports of infeasibility that occur before optimization begins, that is, reports that arise from preprocessing.

Early reports of infeasibility based on preprocessing reductions

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: primal and dual reductions.

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. The *ILOG CPLEX Parameters Reference Manual* documents this parameter: *primal and dual reduction type* (Reduce, CPX_PARAM_REDUCE).

If a model is declared by the preprocessor to be infeasible or unbounded and the user believes that it might be infeasible, the parameter can be set to 1 by the user, and the preprocessor will perform only primal reductions.

If the preprocessor still finds inconsistency in the model, the preprocessor will declare the model 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 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 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

Discusses tactics to diagnose the cause of an unbounded outcome in the optimization of a model and suggests ways to avoid an unbounded outcome.

In this section

What is unboundedness?

Defines unboundedness and tells how to access further information about the problem.

Avoiding unboundedness in a model

Suggests ways to eliminate unboundedness from a model.

Diagnosing unboundedness

Explains typical messages related to unboundedness and suggests ways to access further information.

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 assumes that you first relax any discrete elements, and thus you deal with an unbounded continuous optimization problem, when you try to diagnose the cause.

Note: The reverse of that observation (that an unbounded discrete model necessarily has 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 detected that the model has an *unbounded ray*. That is, for 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 discover 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 in a model

Unboundedness can be viewed as an under-constrained condition; such an outcome may result 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 ILOG 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. In other words, 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 with a worse value of the objective function 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 detected the unboundedness. For example, if the preprocessing 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 detect 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 in a model* is to temporarily assign finite bounds to all variables. By solving the modified model and discovering 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 exists, 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 File Format Reference Manual* for a brief 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* 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

Describes 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.

In this section

What is a conflict?

Defines a conflict as a set of mutually contradictory constraints in a model.

What a conflict is not

Distinguishes conflict refiner from FeasOpt, from data modification, from infeasibility induced by cutoff values.

How to invoke the conflict refiner

Describes routines and methods to invoke the conflict refiner.

How a conflict differs from an IIS

Distinguishes a conflict from an irreducibly inconsistent set (IIS).

Meet the conflict refiner in the Interactive Optimizer

Introduces the conflict refiner in the Interactive Optimizer.

Interpreting conflict

Interprets the conflict detected by the conflict refiner in the Interactive Optimizer.

More about the conflict refiner

Describes the behavior of the conflict refiner in greater detail.

Refining a conflict in a MIP start

Describes how a MIP start may be used with the conflict refiner.

Using the conflict refiner in an application

Describes an application using the conflict refiner in the C++ API.

Comparing a conflict application to Interactive Optimizer

Contrasts the conflict refiner in an application and in the Interactive Optimizer to demonstrate features available only in the Callable Library and Concert Technology, not in the Interactive Optimizer:

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*, 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 the *lower cutoff* and *upper cutoff* 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

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

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</code> <code>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

Introduces the conflict refiner in the Interactive Optimizer.

In this section

Limits of the conflict refiner in the Interactive Optimizer

Summarizes the limits of the conflict refiner in the Interactive Optimizer.

A model for the conflict refiner

Describes a model to exercise the conflict refiner in the Interactive Optimizer.

Optimizing the example

Displays results of optimizing that model in the Interactive Optimizer.

Interpreting the results and detecting conflict

Explains results of the example in the Interactive Optimizer.

Displaying a conflict in the Interactive Optimizer

Displays a conflict detected by the conflict refiner in the Interactive Optimizer.

Limits of 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*.

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 in the Interactive Optimizer

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

Interprets the conflict detected by the conflict refiner in the Interactive Optimizer.

In this section

Understanding the conflict in the model

Describes the reported conflict in terms of the model in the example.

Deleting a constraint

Shows the effect of deleting a constraint from a conflict.

Understanding a conflict report

Explains lower bounds reported in a conflict by the conflict refiner.

Summing equality constraints

Explains sums of constraints reported in a conflict by the conflict refiner.

Changing a bound

Explains the effect of changing a bound reported in a conflict by the conflict refiner.

Adding a constraint

Explains the effect of adding a constraint to an infeasible model.

Changing bounds on cost

Explains the effect of changing bounds on cost in the model.

Relaxing a constraint

Explains the effect of relaxing a constraint in the model.

Understanding the conflict in the model

In those results in *Interpreting the results and detecting conflict*, you 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 managerial 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 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 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 detected 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.  
MIP-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*. 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. One reason that refining a minimal conflict may take longer is that multiple passes (that is, iterations) are performed; each iteration solves a submodel to decide its feasibility status. Another reason is that even if the full model is quickly detected to be infeasible, the infeasibility of the submodels may be less blatant and thus require more time to analyze. It can happen that a number of refinement iterations proceed quickly, and then suddenly no further progress is seen for quite a long time. This deceptive appearance of lack of progress may be especially noticeable in the case of mixed integer models, where a proof of infeasibility may become a quite difficult mathematical problem.

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 discovering 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 >= 400
Bounds
0 <= x1 <= 100
0 <= x2 <= 100
0 <= x3 <= 100
```

the infeasibility could be repaired by one change, namely, by increasing the upper bound of x_3 to be 200. However, with the way the variables are defined, this modification makes little sense. It is more likely that the model contains a subtle mistake in modeling (if the constraint should include more than three variables, for example).

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 signs of instability can be this switching back and forth from feasibility to infeasibility, the user should be alert to this possibility. The

conflict refiner halts and returns 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*.

Refining a conflict in a MIP start

The **conflict refiner** can accept a MIP start. This feature may aid you in debugging a MIP start that CPLEX rejects. That is, you can use the conflict refiner to analyze a rejected MIP start and may then be able to repair the MIP start appropriately.

To use the conflict refiner to debug a rejected MIP start, follow these steps:

1. Add the MIP start to the CPLEX object.
2. Call a special version of the conflict refiner, specifying which MIP start to consider:
 - ◆ In Concert Technology
 - In the C++ API, use `IloCplex::refineMIPStartConflict`.
 - In the Java API, use `IloCplex.refineMIPStartConflict`.
 - In the .NET API, use `Cplex.RefineMIPStartConflict`.
 - ◆ In the Callable Library, use `CPXrefinemipstartconflict` or `CPXrefinemipstartconflictext`.
 - ◆ In the Interactive Optimizer, use the command `conflict i` where `i` specifies the index of the MIP start.

When a MIP start is added to the current model, you may specify an effort level to tell CPLEX how much effort to expend in transforming the MIP start into a feasible solution. The conflict refiner respects effort levels **except** level 1 (one): check feasibility. It does **not** check feasibility of a MIP start.

Using the conflict refiner in an application

Describes an application using the conflict refiner in the C++ API.

In this section

Example: modifying ilomipex2.cpp

Describes modifications of ilomipex2.cpp to demonstrate the conflict refiner.

What belongs in an application to refine conflict

Highlights elements of an application invoking the conflict refiner.

Example: modifying ilomipex2.cpp

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. 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*. 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.

Comparing a conflict application to Interactive Optimizer

Contrasts the conflict refiner in an application and in the Interactive Optimizer to demonstrate features available only in the Callable Library and Concert Technology, not in the Interactive Optimizer:

In this section

Preferences in the conflict refiner

Highlights preferences in the conflict refiner.

Groups in the conflict refiner

Highlights groups in the conflict refiner.

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.
- ◆ 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 designated 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*, 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 : $c8((x2 + x3 + x4 + x5) \leq 0)$

Repairing infeasibilities with FeasOpt

Documents FeasOpt, a feature for repairing infeasibility in a model.

In this section

What is FeasOpt?

Defines FeasOpt, defines preferences, and describes facilities in FeasOpt.

Invoking FeasOpt

Describes routines and methods to invoke FeasOpt.

Specifying preferences

Describes preferences and their effect on bounds and ranges in FeasOpt.

Example: FeasOpt in Concert Technology

Describes an application to use FeasOpt in the C++ API.

What is FeasOpt?

FeasOpt attempts to repair an infeasibility by modifying the model according to *preferences* set by the user. FeasOpt accepts an infeasible model and selectively relaxes the bounds and constraints in a way that minimizes a weighted penalty function that you define. 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 the *lower cutoff* and *upper cutoff* 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 the table *FeasOpt*.

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 **Concert Technology**, 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 is permitted.

Then the code calls `FeasOpt`. If the `FeasOpt` call succeeds, then several lines of output show 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] >= 150
```

into

```
x[0] + x[1] + x[2] >= 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 within 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 practicality into account.

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.

In this section

User-cut and lazy-constraint pools

Documents pools of user-defined cuts and lazy constraints.

Using goals

Documents goals and their role in a branch & cut search.

Using optimization callbacks

Introduces optimization callbacks.

Goals and callbacks: a comparison

Contrasts goals with callbacks.

Advanced presolve routines

Documents the advanced presolve routines, available only in the Callable Library (C API).

Advanced MIP control interface

Documents the advanced MIP control interface.

Parallel optimizers

Documents the ILOG CPLEX parallel optimizers.

User-cut and lazy-constraint pools

Documents pools of user-defined cuts and lazy constraints.

In this section

What are user cuts and lazy constraints?

Defines user cuts; defines lazy constraints.

What are pools of user cuts or lazy constraints?

Defines pools of user cuts or lazy constraints.

Differences between user cuts and lazy constraints

Distinguishes user cuts from lazy constraints.

Limitations on pools in the C API

Describes a limitation on pools in the C API.

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.

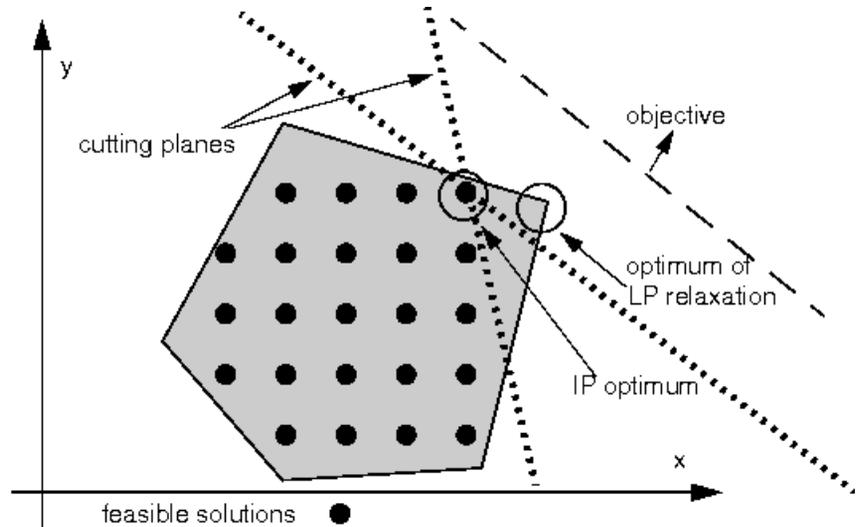
Deleting user cuts and lazy constraints

Describes routines and methods to delete pools of user cuts or lazy constraints.

What are user cuts and lazy constraints?

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.

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.



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*, or
- ◆ by setting up *cut pools* before MIP optimization begins, as explained in *Adding user cuts and lazy constraints*.

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.

Differences between user cuts and lazy constraints

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.

Limitations on pools in the C API

Certain restrictions apply to these pools if you are using the **Callable Library**. (Concert Technology automatically handles these ILOG CPLEX parameter settings for you.) If either of these conditions is violated, ILOG CPLEX issues the error `CPXERR_PRESOLVE_BAD_PARAM` when the MIP optimizer is called.

- ◆ When a user cut pool is present, the *linear reduction switch* (`CPX_PARAM_PRELINEAR` in the Callable Library, `PreLinear` in Concert Technology) must be set to zero.
- ◆ When a lazy constraint pool is present, the *primal and dual reduction type* (`CPX_PARAM_REDUCE` in the Callable Library, `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.

In this section

Using the Component Libraries

Describes routines and methods for adding pools of user cuts or lazy constraints in the APIs.

Using the Interactive Optimizer

Describes display and addition of pools of user cuts and lazy constraints in the Interactive Optimizer.

Reading and writing LP files

Describes reading and writing pools of user cuts or lazy constraints from LP files.

Reading and writing SAV files

Describes reading and writing pools of user cuts or lazy constraints from SAV files.

Reading and writing MPS files

Describes reading and writing pools of user cuts or lazy constraints from MPS files.

Using the Component Libraries

The following facilities add user defined cuts to a user cut pool.

- ◆ The **Callable Library** routine `CPXaddusercuts`
- ◆ The **Concert Technology** methods:
 - `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 **Callable Library** routine is `CPXaddlazyconstraints`.
- ◆ The **Concert Technology** methods
 - `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 appear when you enter the command `display problem all` in the Interactive Optimizer. You can also add user cuts and lazy constraints 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**:
 - `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 are 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**:
 - `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 are 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:

- ◆ *The ROWS section*
- ◆ *User defined cuts in MPS files*
- ◆ *Lazy constraints in MPS files*

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

In the **Callable Library**, the pools of user cut and lazy constraint are cleared by the routines `CPXfreeusercuts` and `CPXfreelazyconstraints`. Clearing the pools does not change the MIP solution.

The **Concert Technology** methods are `clearUserCuts` and `clearLazyConstraints`.

Clearing a pool means that the user cuts and lazy constraints in the pool are removed and are **not** 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

Documents goals and their role in a branch & cut search.

In this section

Branch & cut with goals

Documents the effect of goals in the search.

Special goals in branch & cut

Describes specialized, predefined goals to control the search.

Aggregating goals

Describes aggregate goals (And, Or).

Example: goals in branch & cut

Illustrates goals in a C++ application.

The goal stack

Describes the goal stack.

Memory management and goals

Describes goals in the context of memory management.

Cuts and goals

Distinguishes local cuts from global cuts and illustrates cuts added by a goal

Injecting heuristic solutions

Defines heuristic solutions and describes the injection of a heuristic solution by a goal.

Controlling goal-defined search

Describes node selection strategy with goals.

Example: using node evaluators in a node selection strategy

Illustrates node evaluators in a node selection strategy.

Search limits

Describes search limits induced by a goal.

Branch & cut with goals

Documents the effect of goals in the search.

In this section

What is a goal?

Defines a goal in ILOG CPLEX.

Overview of goals in the search

Describes goals in relation to nodes in the search.

How goals are implemented in branch & cut

Describes implementation of goals in the search in the C++ API.

About the method execute in a goal

Describes the execute method of a goal.

What is a goal?

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 the search*
- ◆ *How goals are implemented in branch & cut*
- ◆ *About the method execute in a goal*

Note: Goals are implemented by `IloCplex::Goal`, not `IloGoal` as in other ILOG products.

Overview of goals in the search

The 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. After 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 the 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 a goal

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 of `execute` is another goal. ILOG CPLEX simply continues by executing this goal.

Special goals in branch & cut

Describes specialized, predefined goals to control the search.

In this section

Or goal

Describes the Or goal to create subnodes of the current node.

And goal

Describes the And goal to combine other goals in a fixed order of execution.

Fail goal

Describes the Fail goal for pruning the search tree.

Local cut goal

Describes a local cut goal to add a cut to a node.

Null goal

Describes the null goal to stop further branching.

Branch as CPLEX goal

Describes a branch goal to return control to ILOG CPLEX.

Solution goal

Describes a goal to inject a solution at a node.

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*.

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*, *Controlling goal-defined search*, and *Search limits*.

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

```
ILOCPLXGOAL1(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`, the 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*.

After 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*, 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 push 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

```
var ≤[val]
```

(where

$\lfloor val \rfloor$

denotes the floor of val) on the goal stack of the first subnode. Similarly, it pushes a local cut goal for

$var \lfloor val \rfloor + 1$

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. That fact 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*.

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::Goal` 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 convention 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, the 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 the 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 the 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*, 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 derive from interpreting the problem as a resource allocation model on five machines with scheduling horizon constraints and transaction times. The first three cuts break symmetries among the machines, while the others capture minimum bounds on transaction costs.

Of course, the best way to solve the `noswot` model with these cuts is 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 to 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 at `IloInfinity` to avoid checking it again unnecessarily.

The global cut goal for `lhs[i] 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 any global cut goals are added, 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` in the Java API or `Cplex.BranchAsCplex` in the .NET API) 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, decide it really is 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 sets 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 node selection 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. Here is the application order:

- ◆ 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). Likewise, in the .NET API, node evaluators are implemented in `Cplex.NodeEvaluator`.

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 this 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 immediately 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` or `Cplex.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 difference is the definition of class `DepthEvaluatorI` as a subclass of `IloCplex::NodeEvaluatorI`. It implements 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 return 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 a high integer infeasibility sum; to choose between nodes with the same integer infeasibility sum, it applies the `DepthEvaluator`. Application of 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` in a new goal `iiSumGoal`. Similarly, the goal `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` or `CplexBranchAsCplex`). 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, the goal `BranchAsCplex` 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 a specified condition applies. Search limits are implemented in subclasses of the 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 optimization callbacks

Introduces optimization callbacks.

In this section

What are callbacks?

Defines callbacks.

Informational callbacks

Documents informational callbacks.

Query or diagnostic callbacks

Documents query or diagnostic callbacks.

Control callbacks

Documents control callbacks.

Implementing callbacks in ILOG CPLEX with Concert Technology

Documents callbacks in Concert Technology.

Example: deriving the simplex callback ilolpex4.cpp

Illustrates derivation of a simplex callback in the C++ API.

Implementing callbacks in the Callable Library

Documents callbacks in the Callable Library C API.

Example: using callbacks lpex4.c

Illustrates callbacks in the C API.

Example: controlling cuts iloadmipex5.cpp

Illustrates callbacks to control cuts in the C++ API.

Interaction between callbacks and ILOG CPLEX parallel optimizers

Describes interaction of callbacks with parallel optimizers.

Return values for callbacks

Documents return values appropriate in user-written callbacks.

Terminating without callbacks

Describes ways to terminate optimization without a callback.

What are 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 or during a tuning session. To use callbacks (either optimization or tuning callbacks) with ILOG CPLEX, you must first write the callback function, and then pass it to ILOG CPLEX.

There are three types of optimization callbacks: informational callbacks, query callbacks, and control callbacks. You will find additional information about callbacks in this manual in *Callbacks for tuning* and in *Advanced MIP control interface*. This topic concentrates on optimization callbacks.

Note: 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::OptimizationCallbackI` is `IloCplex.OptimizationCallback`.

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.OptimizationCallback`.

Informational callbacks

Documents informational callbacks.

In this section

What is an informational callback?

Defines an informational callback.

Reference documents about informational callbacks

Tells where to find reference documents about informational callbacks.

Where to find examples of informational callbacks

Tells where to find sample applications of informational callbacks.

What informational callbacks can return

Tells what informational callbacks return.

What is an informational callback?

An *informational callback* is a user-written routine that enables your application to access information about the current mixed integer programming (MIP) optimization without sacrificing performance and without interfering in the search of the solution space. The algorithms call an informational callback when the algorithm finds it appropriate; for some algorithms, an informational callback is called at every node; for other algorithms, an informational callback is called at convenient points in the progress of the algorithm. *Information returned by informational callbacks* summarizes the information that an informational callback can return.

An informational callback can also enable your application to abort (that is, to terminate) optimization.

Informational callbacks are compatible with MIP dynamic search. For many models, MIP dynamic search finds feasible and optimal solutions more quickly than conventional MIP branch & cut.

Informational callbacks are also compatible with all modes of parallel optimization (if your application is licensed for parallel optimization). For more information about deterministic and opportunistic modes of parallel optimization, see *Determinism of results* and *Parallel MIP optimizer*.

Reference documents about informational callbacks

In Concert Technology, you implement an informational callback as an object of a class.

◆ In the **C++ API**, an informational callback is an instance of `IloCplex::MIPInfoCallback` or one of these derived subclasses:

- `IloCplex::DisjunctiveCutInfoCallback`
- `IloCplex::FlowMIRCutInfoCallback`
- `IloCplex::FractionalCutInfoCallback`
- `IloCplex::ProbingInfoCallback`

An informational callback is installed in a C++ application by the method `IloCplex::use`.

◆ In the **Java API**, an informational callback is an instance of `MIPInfoCallback` or one of these derived subclasses:

- `IloCplex.DisjunctiveCutInfoCallback`
- `IloCplex.FlowMIRCutInfoCallback`
- `IloCplex.FractionalCutInfoCallback`
- `IloCplex.ProbingInfoCallback`

An informational callback is installed in a Java application by the method `IloCplex.use`.

◆ In the **.NET API**, an informational callback is an instance of `Cplex.MIPInfoCallback` or one of these derived subclasses:

- `Cplex.DisjunctiveInfoCallback`
- `Cplex.FlowMIRCutInfoCallback`
- `Cplex.FractionalCutInfoCallback`
- `Cplex.ProbingInfoCallback`

An informational callback is installed in a .NET application by the method `Cplex.Use`.

In the **Callable Library** (C API), use the following routines to install and access an informational callback:

- ◆ `CPXsetinfocallbackfunc`
- ◆ `CPXgetinfocallbackfunc`

Informational callbacks in a C application may call **only** the routines `CPXgetcallbackincumbent` or `CPXgetcallbackinfo`.

Where to find examples of informational callbacks

For examples of how to create and use an informational callback, see the following samples of code in *yourCPLEXhome/examples/src*.

- ◆ mipex4.c
- ◆ ilomipex4.cpp
- ◆ IloMIPex4.java
- ◆ MIPex4.cs
- ◆ MIPex4.vb

What informational callbacks can return

Information returned by informational callbacks

Symbolic name	Meaning
CPX_CALLBACK_INFO_BEST_INTEGER	objective value of the best integer solution
CPX_CALLBACK_INFO_MY_THREAD_NUM	identifying number of the thread where the call of the callback occurred
CPX_CALLBACK_INFO_BEST_REMAINING	objective value of the best remaining node
CPX_CALLBACK_INFO_NODE_COUNT	how many nodes have been processed
CPX_CALLBACK_INFO_NODES_LEFT	how many nodes remain to be processed
CPX_CALLBACK_INFO_MIP_ITERATIONS	how many MIP iterations have occurred
CPX_CALLBACK_INFO_CUTOFF	cutoff information
CPX_CALLBACK_INFO_MIP_FEAS	feasibility information
CPX_CALLBACK_INFO_PROBE_PHASE	which phase of probing (0-3)
CPX_CALLBACK_INFO_PROBE_PROGRESS	progress in probing expressed as a fraction
CPX_CALLBACK_INFO_FRACCUT_PROGRESS	progress in fractional cuts expressed as a fraction
CPX_CALLBACK_INFO_DISJCUT_PROGRESS	progress in disjunctive cuts expressed as a fraction
CPX_CALLBACK_INFO_FLOWMIR_PROGRESS	progress in MIR cuts expressed as a fraction
CPX_CALLBACK_INFO_ENDTIME	time stamp specifying when the optimization will terminate if optimization does not finish before that point

Query or diagnostic callbacks

Documents query or diagnostic callbacks.

In this section

What are query or diagnostic callbacks?

Defines a query or diagnostic callback.

Where query callbacks are called

Tells where query callbacks are called by ILOG CPLEX.

Query callbacks and dynamic search

Describes query callbacks as incompatible with dynamic search.

Query callbacks and deterministic parallel search

Describes query callbacks as incompatible with deterministic parallel search.

What are query or diagnostic callbacks?

Query or diagnostic callbacks allow you to monitor an ongoing optimization, and optionally to abort it (that is, to terminate it). Query callbacks access more detailed information about the current optimization than do informational callbacks. As a side effect, query or diagnostic callbacks may slow progress. Furthermore, query or diagnostic callbacks make assumptions about the traversal of a conventional branch & cut tree; those assumptions about a mixed integer program (MIP) may be incorrect during during dynamic search or during deterministic search in parallel optimization. For more detail about this point, see *Query callbacks and dynamic search* and *Query callbacks and deterministic parallel search*.

Where query callbacks are called

Query or diagnostic callbacks are distinguished by the place where they are called during an optimization. There are nine such places where ILOG CPLEX calls a query or diagnostic callback:

- ◆ The presolve query callback is called regularly during presolve.
 - `IloCplex::PresolveCallbackI` in the C++ API
 - `IloCplex.PresolveCallback` in the Java API
 - `Cplex.PresolveCallback` in the .NET API
 - `CPXsetlpcallbackfunc` in the Callable Library (C API)
- ◆ The crossover query callback is called regularly during crossover from a barrier solution to a simplex basis.
 - `IloCplex::CrossoverCallbackI` in the C++ API
 - `IloCplex.CrossoverCallback` in the Java API
 - `Cplex.CrossoverCallback` in the .NET API
 - `CPXsetlpcallbackfunc` in the Callable Library (C API)
- ◆ The network query callback is called regularly during the network simplex algorithm.
 - `IloCplex::NetworkCallbackI` in the C++ API
 - `IloCplex.NetworkCallback` in the Java API
 - `Cplex.NetworkCallback` in the .NET API
 - `CPXsetnetcallbackfunc` in the Callable Library (C API)
- ◆ The barrier query callback is called at each iteration during the barrier algorithm.
 - `IloCplex::BarrierCallbackI` or `IloCplex::ContinuousCallbackI` in the C++ API
 - `IloCplex.BarrierCallback` or `IloCplex.ContinuousCallback` in the Java API
 - `Cplex.BarrierCallback` or `Cplex.ContinuousCallback` in the .NET API
 - `CPXsetlpcallbackfunc` in the Callable Library (C API)
- ◆ The simplex query callback is called at each iteration during the simplex algorithm.

- `IloCplex::SimplexCallbackI` or `IloCplex::ContinuousCallbackI` in the C++ API
- `IloCplex.SimplexCallback` or `IloCplex.ContinuousCallback` in the Java API
- `Cplex.SimplexCallback` or `Cplex.ContinuousCallback` in the .NET API
- `CPXsetlpcallbackfunc` in the Callable Library (C API)
- ◆ The MIP query callback is called regularly during the branch and cut search.
 - `IloCplex::MIPCallbackI` in the C++ API
 - `IloCplex.MIPCallback` in the Java API
 - `Cplex.MIPCallback` in the .NET API
 - `CPXsetmipcallbackfunc` in the Callable Library (C API)
- ◆ The probing query callback is called regularly during probing.
 - `IloCplex::ProbingCallbackI` in the C++ API
 - `IloCplex.ProbingCallback` in the Java API
 - `Cplex.ProbingCallback` in the .NET API
- ◆ The fractional cut query callback is called regularly during the generation of fractional cuts.
 - `IloCplex::FractionalCutCallbackI` in the C++ API
 - `IloCplex.FractionalCutCallback` in the Java API
 - `Cplex.FractionalCutCallback` in the .NET API
- ◆ The disjunctive cut query callback is called regularly during the generation of disjunctive cuts.
 - `IloCplex::DisjunctiveCutCallbackI` in the C++ API
 - `IloCplex.DisjunctiveCutCallback` in the Java API
 - `Cplex.DisjunctiveCutCallback` in the .NET API

In the C++ API, a query callback is installed by the method `use`.

In the Java API, a query callback is installed by the method `IloCplex.use`.

In the .NET API, a query callback is installed by the method `Cplex.Use`.

In Callable Library applications (C API), a query callback is a user-written function installed by the routine `CPXsetmipcallbackfunc`, `CPXsetlpcallbackfunc`, or `CPXsetnetcallbackfunc`.

Query callbacks and dynamic search

Query or diagnostic callbacks are not compatible with dynamic search, a feature which explores a solution space in a way that departs from a conventional branch & cut tree.

Normally, ILOG CPLEX chooses whether to apply dynamic search or conventional branch & cut based on characteristics of the model. To benefit from dynamic search, a MIP must not include query callbacks. In the presence of query or diagnostic callbacks, ILOG CPLEX turns off dynamic search, issues a warning, and applies conventional branch & cut.

If you want to avoid this warning in an application where query or diagnostic callbacks are present, you can deliberately turn off dynamic search yourself by setting the *MIP dynamic search switch* (MIPSearch, CPX_PARAM_MIPSEARCH) to 1 (one).

Query callbacks and deterministic parallel search

Query or diagnostic callbacks are not compatible with deterministic parallel search, a feature which explores a solution space in a way that departs from a conventional branch & cut tree.

In the presence of a query or diagnostic callback, ILOG CPLEX turns off deterministic parallel search and invokes opportunistic parallel search instead, if your application is licensed for parallel MIP optimization.

For more information about deterministic and opportunistic parallel MIP optimization, see *Determinism of results* and *Parallel MIP optimizer*.

Control callbacks

Documents control callbacks.

In this section

What are control callbacks?

Defines control callbacks.

What control callbacks do

Describes what control callbacks do.

Control callbacks and dynamic search

Describes control callbacks as incompatible with dynamic search.

Control callbacks and deterministic parallel search

Describes control callbacks as incompatible with deterministic parallel search.

What are control callbacks?

Control callbacks allow you to control the branch & cut search during the optimization of MIP problems. Because control callbacks intervene in the search, the presence of a control callback in an application will cause ILOG CPLEX to turn off dynamic search. Likewise, the presence of a control callback will cause ILOG CPLEX to turn off deterministic parallelism, if the application is licensed for parallelism. *Control callbacks and dynamic search* and *Control callbacks and deterministic parallel search* offer more information about this point.

If you want to take advantage of dynamic search or deterministic parallelism in your application, and you see a need for a callback to report progress, consider an informational callback instead of a control callback. *Informational callbacks* explains more about this alternative, which is compatible with dynamic search and with deterministic parallelism.

What control callbacks do

If you determine that your application needs to seize control, intervene in the search, and redirect the optimizer, then the following control callbacks are available to do so.

- ◆ The node callback allows you to query and optionally overwrite the next node ILOG CPLEX will process during a branch & cut search.
 - `IloCplex::NodeCallbackI` in the C++ API
 - `IloCplex.NodeCallback` in the Java API
 - `Cplex.NodeCallback` in the .NET API
 - `CPXsetnodecallbackfunc` in the Callable Library (C API)
- ◆ The solve callback allows you to specify and configure the optimizer option to be used for solving the LP at each individual node.
 - `IloCplex::SolveCallbackI` in the C++ API
 - `IloCplex.SolveCallback` in the Java API
 - `Cplex.SolveCallback` in the .NET API
 - `CPXsetsolvecallbackfunc` in the Callable Library (C API)
- ◆ The cut callback allows you to add problem-specific cuts at each node.
 - `IloCplex::CutCallbackI` in the C++ API
 - `IloCplex.CutCallback` in the Java API
 - `Cplex.CutCallback` in the .NET API
 - `CPXsetcutcallbackfunc` in the Callable Library (C API)
- ◆ 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.
 - `IloCplex::HeuristicCallbackI` in the C++ API
 - `IloCplex.HeuristicCallback` in the Java API
 - `Cplex.HeuristicCallback` in the .NET API
 - `CPXsetheuristiccallback` in the Callable Library (C API)
- ◆ The branch callback allows you to query and optionally overwrite the way ILOG CPLEX will branch at each node.
 - `IloCplex::BranchCallbackI` in the C++ API

- `IloCplex.BranchCallback` in the Java API
- `Cplex.BranchCallback` in the .NET API
- `CPXsetbranchcallbackfunc` in the Callable Library (C API)
- ◆ The incumbent callback allows you to check and optionally reject incumbents found by ILOG CPLEX during the search.
 - `IloCplex::IncumbentCallbackI` in the C++ API
 - `IloCplex.IncumbentCallback` in the Java API
 - `Cplex.IncumbentCallback` in the .NET API
 - `CPXsetincumbentcallbackfunc` in the Callable Library (C API)

Control callbacks and dynamic search

Control callbacks are not compatible with dynamic search, a feature which explores a solution space in a way that departs from a conventional branch & cut tree.

Normally, ILOG CPLEX chooses whether to apply dynamic search or conventional branch & cut based on characteristics of the model. To benefit from dynamic search, a MIP must not include control callbacks. In the presence of control callbacks, ILOG CPLEX turns off dynamic search, issues a warning, and applies conventional branch & cut.

If you want to avoid this warning in an application where control callbacks are present, you can deliberately turn off dynamic search yourself by setting the *MIP dynamic search switch* (MIPSearch, CPX_PARAM_MIPSEARCH) to 1 (one).

Control callbacks and deterministic parallel search

Control callbacks are not compatible with deterministic parallel search, a feature which explores a solution space in a way that departs from a conventional branch & cut tree.

In the presence of a control callback, ILOG CPLEX turns off deterministic parallel search and invokes opportunistic parallel search instead, if your application is licensed for parallel MIP optimization.

For more information about deterministic and opportunistic parallel MIP optimization, see *Determinism of results* and *Parallel MIP optimizer*.

Implementing callbacks in ILOG CPLEX with Concert Technology

Documents callbacks in Concert Technology.

In this section

How callback classes are organized

Describes the hierarchy of callback classes.

Writing callback classes by hand

Describes user's implementation of a callback class.

Writing callbacks with macros in C++

Describes implementation of a callback class by means of a macro.

Callback interface

Describes the common callback interface.

The continuous callback

Describes the common interface of callbacks for simplex and barrier optimizers.

How callback classes are organized

Callbacks are accessed via the `IloCplex::Callback` handle class in the C++ implementation of `IloCplex`. An instance of that handle class 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` or `Cplex.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* or *Graph* 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 ILOG CPLEX: 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 in C++*.

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** applications; instead, `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 remove all callbacks by calling the method `IloCplex.clearCallbacks` or `CplexClearCallbacks`. 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 in C++

This is how to implement a callback using macros. Since macros are not supported in Java nor in .NET, this technique applies only to C++ applications.

Start by deciding 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`. *Callback macros* lists the callbacks and the corresponding macros and classes (where `n` is a placeholder for 0 to 7).

Callback macros

Callback	Macro	Class
presolve	<code>ILOPRESOLVECALLBACKn</code>	<code>IloCplex::PresolveCallbackI</code>
continuous	<code>ILOCONTINUOUSCALLBACKn</code>	<code>IloCplex::ContinuousCallbackI</code>
simplex	<code>ILOSIMPLEXCALLBACKn</code>	<code>IloCplex::SimplexCallbackI</code>
barrier	<code>ILOBARRIERCALLBACKn</code>	<code>IloCplex::BarrierCallbackI</code>
crossover	<code>ILOCROSSOVERCALLBACKn</code>	<code>IloCplex::CrossoverCallbackI</code>
network	<code>ILONETWORKCALLBACKn</code>	<code>IloCplex::NetworkCallbackI</code>
MIP info	<code>ILOMIPINFOCALLBACKn</code>	<code>IloCplex::MIPInfoCallbackI</code>
MIP	<code>ILOMIPCALLBACKn</code>	<code>IloCplex::MIPCallbackI</code>
probing info	<code>ILOPROBINGINFOCALLBACKn</code>	<code>IloCplex::ProbingCallbackI</code>
probing	<code>ILOPROBINGCALLBACKn</code>	<code>IloCplex::ProbingCallbackI</code>
fractional cut info	<code>ILOFRACTIONALCUTINFOCALLBACKn</code>	<code>IloCplex::FractionalCutCallbackI</code>
fractional cut	<code>ILOFRACTIONALCUTCALLBACKn</code>	<code>IloCplex::FractionalCutCallbackI</code>
disjunctive cut info	<code>ILODISJUNCTIVECUTINFOCALLBACKn</code>	<code>IloCplex::DisjunctiveCutCallbackI</code>
disjunctive cut	<code>ILODISJUNCTIVECUTCALLBACKn</code>	<code>IloCplex::DisjunctiveCutCallbackI</code>

Callback	Macro	Class
flow MIR cut info	ILOFLOMIRCUTINFOCALLBACKn	IloCplex::FlowMIRCutInfoCallbackI
flow MIR cut	ILOFLOMIRCUTCALLBACKn	IloCplex::FlowMIRCutCallbackI

The protected methods of the corresponding class and its base classes specify the functions that can be called for implementing your callback. See the *ILOG CPLEX Reference Manual* with respect to these classes for details of which functions can be called.

Here is an example of how to implement a simplex callback with the name `MyCallback` that takes one argument:

```
virtual IloCplex::CallbackI*
IloCplex::CallbackI::duplicateCallback() const = 0;

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 instance of `IloCplex` declared as `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 convention allows you to call `mycallback.end` 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.Callback or Cplex.Callback)`. 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 common interface consists of these 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),
- ◆ `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, no manipulation of any extracted modeling object is allowed during the execution of a callback. No modification is allowed of any array or expression not local to the callback function itself (that is, constructed and `end`-ed in it). The only exception is the modification of array elements. For example, `x[i] = 0` would be permissible, whereas `x.add(0)` would **not** unless `x` is a local array of the callback.

The continuous callback

The second special callback class is `IloCplex::ContinuousCallbackI` (`IloCplex.ContinuousCallback` or `Cplex.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 methods `getNIterations`, `isFeasible`, `getObjValue`, and `getInfeasibility` are methods provided in the base class of the callback, `IloCplex::ContinuousCallbackI` (`IloCplex.ContinuousCallback` or `Cplex.ContinuousCallback`). See the *ILOG CPLEX Reference Manual* for the complete list of methods provided for each callback class.

Here is the previous sample of code, with the macro `ILOSIMPLEXCALLBACK0` 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*. 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:

- ◆ `cplex.use(MyCallback(env));` in C++
- ◆ `cplex.use(new MyCallback());` in Java
- ◆ `cplex.Use(new MyCallback());` in .NET

In the case of C++, the function `MyCallback` creates an instance of the implementation class `MyCallbackI`. A handle to this implementation object is passed to the method `IloCplex::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 manage one callback for each callback class at the same time. For example, a simplex callback and a MIP callback can be used at the same time.

The complete program, `ilolpex4.cpp`, appears online in the standard distribution at *yourCPLEXinstallation/examples/src*. In the same location, there are also samples in Java (`LPex4.java`) and in the .NET API (`LPex4.cs` and `LPex4.vb`).

Implementing callbacks in the Callable Library

Documents callbacks in the Callable Library C API.

In this section

Callable Library callback facilities

Describes features to support callbacks in the C API.

Setting callbacks

Describes organization of callbacks in the C API.

Callbacks for continuous and discrete problems

Describes when callbacks are invoked and enumerates the arguments of a user-defined callback in the C API.

Callable Library callback facilities

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, query or 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 `wherefrom` 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 periodically 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`, specifies 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`.

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 manual *ILOG CPLEX Getting Started*. 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 *messages to screen switch* `CPX_PARAM_SCRIND`) is not turned on. Only the callback function produces output. Consequently, this program calls `CPXgeterrorstring` to retrieve 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. *Status variables in lpex4.c* summarizes those status variables.

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*.

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 and MIPLIB 2003 test-sets, 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 still substantial. 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 were 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:

```

ILOUTCALLBACK3(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 a handle for it, an instance of `IloCplex::Callback`.

As specified by the 3 in the macro name, the constructor of `CtCallbackI` takes three arguments: `lhs`, `rhs`, and `eps` (lefthand side, righthand side, and tolerance). The constructor stores them as private members to have direct access to them in the callback function, implemented as the 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::LazyConstraintCallbackI {
    IloExprArray lhs;
    IloNumArray rhs;
    IloNum eps;
public:
    IloCplex::CallbackI* duplicateCallback() const {
        return (new(getEnv()) CtCallbackI(*this));
    }
    CtCallbackI(IloEnv env, IloExprArray xx1, IloNumArray xx2, IloNum xx3) :
        IloCplex::LazyConstraintCallbackI(env, lhs(xx1), rhs(xx2), eps(xx3))
    {
    }
    void main();
};
IloCplex::Callback
CtCallback(IloEnv env, IloExprArray lhs, IloNumArray rhs, IloNum eps) {
    return (IloCplex::Callback(new(env) CtCallbackI(env, 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 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 round-off. 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] - rhs[i]`. It is added to the LP by 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, the 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 the cut itself is created or when the application tries 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 re-thrown. Re-throwing 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 at `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`. Variables are identified by querying their names. After 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 this line:

```
cplex.use(CtCallback(env, lhs, rhs, cplex.getParam(IloCplex::EpRHS)));
```

The function `CtCallback` constructs an instance of our callback class `CtCallbackI` and returns a handle (an instance of `IloCplex::Callback`) for it. This handle is directly passed to the 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 the tolerance of `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 `noswot` cuts by calling.

`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.use(new Callback(makeCuts(cplex, lp)));
cplex.addCuts(makeCuts(var));

```

as shown in the example `iloadmipex4.cpp` in the distribution. During branch & cut, ILOG CPLEX considers 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 approach or adding the cuts directly to the model gives better performance during the solution of 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`.

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. (You should not change parameters from within a callback.) 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.

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 terminates. 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`, specifying that the process stopped because the user intervened..

For both LP, QP, QCP, and MIP problems, if the LP/QP/QCP callback returns a nonzero value during 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 terminates, and the status returned by `IloCplex::getStatus` or `CPXgetstat` is one of the values in the table *Status of nonzero callbacks for MIPs*.

Status of nonzero callbacks for MIPs

Symbolic constant	Meaning
<code>CPXMIP_ABORT_FEAS</code>	current solution integer feasible
<code>CPXMIP_ABORT_INFEAS</code>	no integer feasible solution found

Terminating without callbacks

If you simply want to terminate optimization under circumstances defined by your application, then you do not necessarily need to write a callback to do so. Instead, you can invoke termination by these means:

- ◆ In the C++ API, pass an instance of the class `IloCplex::Aborter` to an instance of `IloCplex`. Then call the method `IloCplex::Aborter::abort` to terminate optimization.
- ◆ In the Java API, pass an instance of the class `IloCplex.Aborter` to an instance of `IloCplex`. Then call the method `IloCplex.Aborter.abort` to terminate optimization.
- ◆ In the .NET API, pass an instance of the class `Cplex.Aborter` to an instance of `Cplex`. Then call the method `Cplex.Aborter.Abort` to terminate optimization.
- ◆ In the Callable Library (C API), call the routine `CPXsetterminate` to set a pointer to the termination signal. Initially, the value of the termination signal should be zero. When your application sets the termination signal to a nonzero value, then ILOG CPLEX will terminate optimization.

Goals and callbacks: a comparison

Contrasts goals with callbacks.

In this section

Overview

Outlines advantages of goals and callbacks.

Overview

Goals and callbacks both provide an API within ILOG CPLEX 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 goals and callbacks. In fact, the goal API is built on top of callbacks. As a consequence, you can **not** use callbacks and goals at the same time. To help you choose which API is more suited to your needs, this topic examines commonalities and differences between both.

A quick checklist comparing goals and callbacks

Both approaches (goals and callbacks) allow you to control the branch & cut search used by ILOG CPLEX 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 `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.

Further notes about goals and callbacks

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.

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.

Example contrasting goals and callbacks

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. The choice of the appropriate branch may be quite difficult for constraints not modeled with linear expressions, even though ILOG CPLEX supports branching on hyperplanes.

Advanced presolve routines

Documents the advanced presolve routines, available only in the Callable Library (C API).

In this section

Introduction to presolve

Describes presolve with an example contrasting continuous and discrete optimization.

A proposed example

Describes special considerations about presolve.

Restricting presolve reductions

Describes conditions under which a user might restrict presolve.

Manual control of presolve

Describes features to control presolve.

Modifying a problem

Describes conditions in which to modify a model after presolve.

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 unpresolves the solution to produce a solution to the original problem. During this process, presolve builds an entirely new ‘presolved’ problem and stores enough information to translate a solution to this problem back to a solution to the original problem. This information is hidden within the user's problem (in the CPLEX LP problem object, for Callable Library users) and was inaccessible to the user in CPLEX releases prior to 7.0.

The presolve process for a mixed integer program is similar, but has a few important differences. First, the actual presolve reductions differ. Integrality restrictions allow CPLEX to perform several classes of reductions that are invalid for continuous variables. A second difference is that the MIP solution process involves a series of linear program solutions. In the MIP branch & cut tree, a linear program is solved at each node. MIP presolve is performed at the beginning of the optimization and applied a second time to the root relaxation, unless

the *relaxed LP presolve switch* (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 model. Again, presolve stores the presolved model 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 model. 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 model useless for the re-optimization. (There is an example in *Restricting presolve reductions*.) 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 model or perform a new presolve on the modified problem. In the former case, the re-optimization 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 model, so new constraints can be added to the presolved model. In short, it provides a variety of capabilities.

When processing 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 model) 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*.

Restricting presolve reductions

Describes conditions under which a user might restrict presolve.

In this section

When to alert presolve to modifications

Describes conditions in which a user alerts presolve to modifications of the model.

Adding constraints to the first solution

Describes the effect of adding constraints after solution.

Primal and dual considerations in presolve reductions

Describes modifications that impact primal and dual formulations.

Cuts and presolve reductions

Describes presolve reductions influenced by cuts.

Infeasibility or unboundedness in presolve reductions

Describes special considerations of presolve with respect to infeasible or unbounded models.

Protected variables in presolve reductions

Describes the interaction of protected variables and presolve.

When to alert presolve to modifications

As mentioned in *Introduction to presolve*, 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 apply only to linear programs. Any modifications of QP or QCP models will cause ILOG CPLEX to discard the presolved model.

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 $y_1 \leq -1$
$x_2 + x_3 \leq 5$	$y_1 + y_2 \leq 1$
$0 \leq x_1, x_2, x_3$	$2y_1 + y_2 \leq 1$
	$0 \leq y_1, y_2, y_3$

Note that the first constraint in the dual ($y_1 \leq -1$) 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 model. 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 $y_1 \leq -5y_2 - 1$
$x_2 + x_3 \leq 5$	$y_1 + y_2 + y_3 \leq 1$
$-5x_1 + x_2 \leq 0$	$2y_1 + y_2 \leq 1$
$0 \leq x_1, x_2, x_3$	$0 \leq y_1, y_2, y_3$

Our goal is to add the appropriate constraint to the presolved model and re-optimize. 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 model to re-optimize.

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 *primal and dual reduction type* (`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 *primal and dual reduction type* (`CPX_PARAM_REDUCE`) parameter has one additional effect on the optimization. Normally, the presolved model 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 model 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 model and reclaim the associated memory. The presolved model 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*.)

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 model. 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 model. Set the *linear reduction switch* `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 the *primal and dual reduction type* (CPX_PARAM_REDUCE) parameter, presolve can only conclude that a problem is infeasible and/or unbounded. If the CPX_PARAM_REDUCE parameter 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 .

Note: This 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 model). 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. The routines mentioned here are documented in detail, including arguments and return values, in the *ILOG CPLEX Callable Library Reference Manual*.

The first control function provided by the advanced presolve interface is `CPXpresolve`, which manually invokes presolve on the supplied problem. After this routine is called on a problem, the original problem has a *presolved model* associated with it. Subsequent calls to optimization routines (`CPXprimopt`, `CPXdualopt`, `CPXbaropt`, `CPXmipopt`) will use this presolved model without repeating the presolve, provided no operation that discards the presolved model is performed in the interim. The presolved model is automatically discarded if a problem modification is performed that is incompatible with the setting of the parameter `CPX_PARAM_REDUCE`, documented in *primal and dual reduction type*. (Further information about this point appears in *Modifying a problem*.)

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 model. 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 of the presolved model. 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, then calling `CPXpresolve` to create the new presolved model, then calling `CPXcopystart`, passing the original model and any combination of primal and dual solutions. With nondefault settings of the parameter `CPX_PARAM_REDUCE` *primal and dual reduction type*, 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.

There are subtleties associated with using `CPXcopystart` to start an optimization from an advanced, presolved solution. This routine creates a presolved solution vector only if the presolved model 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.

You can set the *advanced start switch* (`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 branch & cut. After `CPXpresolve` has been called, the user can then call `CPXgetprestat` to obtain information about the reductions performed on the problem. This routine provides information, for each variable in the original problem, about whether the variable was fixed and removed, aggregated out, removed for some other reason, or is still present in the reduced model. It also gives information, for each row in the original problem, about whether it was removed due to redundancy, aggregated out, or is still present in the reduced model. For those rows and columns that are present in the reduced model, this function provides a mapping from original row/column number to row/column number in the reduced model, and vice-versa.

Another situation where a user might want to use `CPXpresolve` is to work directly on the presolved model. By calling `CPXgetredlp` immediately after `CPXpresolve`, the user can obtain a pointer to the presolved model. As an example of how this pointer might be used, the user could call routines `CPXcrushx` and `CPXcrushpi` to presolve primal and dual solution vectors, then call `CPXgetredlp` to get access to the presolved model, then use `CPXcopystart` to copy the presolved solutions into the presolved model, then optimize the problem, and finally call the routines `CPXuncrushx` and `CPXuncrushpi` (`CPXqpuncrushpi` for QPs) to unresolve solutions from the presolved model, creating solutions for the original problem.

The routine `CPXgetredlp` provides the user access to internal CPLEX data structures. The presolved model 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 model (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 model. The most likely use of this routine is to add user cuts to the presolved model during a mixed integer optimization. The advanced presolve interface also provides the reverse operation. The routine `CPXuncrushform` translates a linear expression in the presolved model 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 routine detects 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 model.

The interface allows the user to manually free the memory associated with the presolved model using the routine `CPXfreepresolve`. The next optimization call (or call to `CPXpresolve`) recreates the presolved model.

Modifying a problem

This section briefly discusses the mechanics of modifying a model 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 specify through the parameter `CPX_PARAM_REDUCE`, documented in *primal and dual reduction type*, the types of modifications that are going to be performed on the model. 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 model. These changes are made through the standard problem modification interface (`CPXaddcols`, `CPXchgrhs`, and `CPXchgbd`s).

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 (`CPXchgbd`s). The addition of constraints or changes to the objective value must be done through the two routines `CPXpreaddrows` and `CPXprechgobj`. 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`, `ILOG CPLEX` discards the presolved model, and presolve is re-invoked at the next optimization call. Similarly, `CPLEX` discards the presolved model 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 model. 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 model, consider changing the bounds on the original model. `CPLEX` will discard the presolved model, but calling `CPXpresolve` will cause `CPLEX` to apply presolve to the modified model, with the added benefit of reductions based on the latest 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

Documents the advanced MIP control interface.

In this section

Introducing the advanced MIP control interface

Describes the context and prerequisites of the advanced MIP control interface.

Introducing MIP control callbacks

Documents MIP control callbacks

Heuristic callback

Describes the heuristic callback.

Cut callback

Describes special considerations of the cut callback.

Branch selection callback

Describes special considerations about the branch selection callback.

Incumbent callback

Describes special considerations about the incumbent callback.

Node selection callback

Describes special considerations about the node selection callback.

Solve callback

Describes special considerations about the solve callback.

Introducing the advanced MIP control interface

In this manual, *Using optimization callbacks* introduces callbacks, their purpose, and conventions. Continuing from there, this topic documents the CPLEX advanced MIP control interface, describing control callbacks in greater detail. It assumes that you are already familiar with that introduction to callbacks in general.

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 nondefault parameter settings or priority orders, have failed. See *Troubleshooting MIP performance problems* 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. It's a good idea to become familiar with *Advanced presolve routines* before reading this topic.

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*.

Introducing MIP control callbacks

Documents MIP control callbacks

In this section

What are MIP control callbacks?

Describes MIP control callbacks.

Thread safety and MIP control callbacks

Describes control callbacks in parallel MIP.

Presolve and MIP control callbacks

Describes presolve considerations for MIP control callbacks.

What are MIP control 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 those important steps (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 control callback mechanism to allow the user to do this. If the user installs a control callback routine, CPLEX calls this routine during the branch & cut process to allow the user to intervene.

Thread safety and MIP control callbacks

When you use a control callback with parallel MIP, you must observe several points about default parameter settings, thread-safety, and parallelism. This section addresses those points.

The presence of a control callback in an application with default parameter settings turns off parallel MIP. (Informational callbacks do not have this effect of turning off parallel MIP optimization by default. For more about informational callbacks, see *Informational callbacks*.) In order to continue using parallelMIP in the presence of a control callback, the user must set the *global default thread count* parameter (`Threads`, `CPX_PARAM_THREADS`) to a value strictly greater than one. It is then the responsibility of the user to implement that control callback in such a way that the callback is thread-safe. In particular, the control callback must be written in such a way that it does not depend on the order in which callbacks are called, as no fixed order of calling the callbacks can be guaranteed by parallel ILOG CPLEX.

With respect to determinism, if the user sets the *parallel mode switch* (`ParallelMode`, `CPX_PARAM_PARALLELMODE`) to 1 (one) thus invoking deterministic parallel MIP optimization, it is up to the user to make sure that the control callback does not interfere with the search in any way that would make the search opportunistic. To make sure of that condition, the control callback must use only information queried from ILOG CPLEX itself within the callback as the basis for algorithmic decisions. In other words, no information that accumulated in an external data structure over several invocations of the control callback can be used.

Presolve and MIP control callbacks

This section addresses an important issue related to presolve that the user of MIP control callbacks 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 the *MIP callback switch between original model and reduced, presolved model* (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. After 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 *MIP callback switch between original model and reduced, presolved model*, `CPX_PARAM_MIPCBREDLP`). This pointer can be used to obtain various information about the problem (variable types, etc.), or as an argument for the advanced presolve interface if the user wishes to manually translate between presolved and unpresolved values. In addition, the callback can use the `cbdata` parameter passed to it, along with routine `CPXgetcallbacknodelp`, to obtain a pointer to the node relaxation LP. This can be used to access desired information about the relaxation (row count, column count, etc.). Note that in both cases, the user should never use the pointers obtained from these callbacks to modify the associated problems.

As noted earlier, the `CPX_PARAM_MIPCBREDLP` parameter influences the arguments to the user callback routine. If this parameter is set to its default value of `CPX_ON (1)`, the solution vector returned to the callback, and any feasible solutions returned by the callback, are presolved vectors. They contain one value for each variable in the presolved problem. The same is true of the various callback support routines (`CPXgetcallbackgloballb`, etc.). If the parameter is set to `CPX_OFF (0)`, all these vectors relate to variables of the original problem. Note that this parameter should not be changed in the middle of an optimization.

The user should be aware that the branch & cut process works with the presolved problem, so the code will incur some cost when translating from presolved to original values. This cost is usually small, but can sometimes be significant.

We should also note that if a user wishes to solve linear programs as part of a heuristic callback, the user must make a copy of the node LP (for example, using `CPXcloneprob`). The user should not modify the CPLEX node LP.

Cut callback

The next example to 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 after 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 a call to `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” that 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 *linear reduction switch* (`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 *primal and dual reduction type* parameter, `CPX_PARAM_REDUCE`). 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 nonlinear reductions then will be turned off automatically.

One scenario that merits special attention is when the user knows a large set of cuts because of prior knowledge. Rather than adding them to the original problem, the user may instead wish to add them only when they are 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 the *MIP callback switch between original model and reduced, presolved model*, `CPX_PARAM_MIPCBREDLP`, to `CPX_OFF` in order 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*.

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 *MIP callback switch between original model and reduced, presolved model*, `CPX_PARAM_MIPCBREDLP`, 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 changes an argument 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 *MIP callback switch between original model and reduced, presolved model*, `CPX_PARAM_MIPCBREDLP`, 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 to 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` shows how this callback might be used.

Note: The most common use of this callback is to craft a custom 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

Documents the ILOG CPLEX parallel optimizers.

In this section

What are parallel optimizers?

Describes the parallel optimizers available in ILOG CPLEX.

Threads

Documents threads in the context of parallel optimizers.

Determinism of results

Defines determinism and describes its effect in parallel optimization.

Using parallel optimizers in the Interactive Optimizer

Describes parallel optimization in the Interactive Optimizer.

Using parallel optimizers in the ILOG CPLEX Component Libraries

Describes parallel optimization in the Component Libraries.

Parallel barrier optimizer

Describes the parallel barrier optimizer.

Concurrent optimizer

Describes the concurrent optimizer.

Parallel MIP optimizer

Describes the parallel MIP optimizer.

Clock settings and time measurement

Describes reporting of time in parallel MIP optimization.

What are parallel optimizers?

ILOG CPLEX offers Parallel Barrier, Parallel MIP, and Concurrent optimizers. These parallel optimizers are implemented to run on hardware platforms with parallel processors. These parallel optimizers can be called from the Interactive Optimizer and the Component Libraries.

Threads

Documents threads in the context of parallel optimizers.

In this section

Licensing and threads parameter

Describes the effect of licensing conditions on number of threads in parallel.

Example: threads and licensing

Illustrates interaction of threads and licensing with an example.

Threads and performance considerations

Describes performance consideration for threads.

Licensing and threads parameter

The ILOG CPLEX parallel optimizers are licensed for a specific maximum number of threads. You manage the number of threads that ILOG CPLEX uses with the *global default thread count* parameter (Threads, CPX_PARAM_THREADS) documented in the *ILOG CPLEX Parameters Reference Manual*. At its default setting 0 (zero), the number of threads that ILOG CPLEX actually uses during a parallel optimization is the smaller of:

- ◆ the number of CPU cores available on the computer where ILOG CPLEX is running;
- ◆ the number of threads specified by the license.

If you set the threads parameter to a value greater than its default of 0 (zero), then the number of threads that ILOG CPLEX uses equals that value. When you set the threads parameter to 1 (one), you enforce sequential operation, that is, processing on one thread only. The maximum possible setting is the smaller of:

- ◆ the number of CPU cores available on the computer where ILOG CPLEX is running;
- ◆ the number of threads specified by the license.

The number of threads used by a parallel ILOG CPLEX optimizer is separate from and independent of the number of licensed users. 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, consider the following results:

```
Root node processing (before b&c):
  Real time           =    3.22
Parallel b&c, 2 threads:
  Real time           =    8.29
  Sync time (average) =    2.74
  Wait time (average) =    2.48
  -----
Total (root+branch&cut) =  11.51 sec.
```

When the wait time is particularly high, as in that example, consider changing the parallel mode from deterministic to opportunistic or changing from parallel to sequential processing. *Determinism of results* explains more about these choices.

Another key consideration in setting optimizer and global thread limits is your management of overall system load.

Determinism of results

By default, CPLEX will employ parallel algorithms only as long as the optimization remains deterministic. In this context, *deterministic* means that repeated solving of the same model with the same parameter settings on the same computing platform will follow exactly the same solution path, yielding the same level of performance and the same values in the solution.

The parallel barrier optimizer can not work deterministically because of design considerations. Likewise, by design, the concurrent optimizer cannot work deterministically. (Even though determinism cannot be guaranteed theoretically for parallel barrier or concurrent optimization, these optimizers rarely exhibit behavior that is not deterministic in practice.) Because of this consideration about determinism in their design, and because of its implicit contract to use parallel algorithms deterministically by default, ILOG CPLEX does not invoke these optimizers by default in parallel. Thus, in order to use parallel processors for concurrent optimization or for barrier optimization, you must explicitly set the *global default thread count* parameter (Threads, CPX_PARAM_THREADS) to a value strictly greater than one. This higher setting implicitly specifies to ILOG CPLEX that the user is willing to accept behavior that may not be deterministic.

In contrast, for mixed integer programming (MIP), two parallel implementations are available in ILOG CPLEX: deterministic and opportunistic. Opportunistic parallel optimization requires less synchronization between threads and thus offers better performance on average. Consequently, during development of an application, you may find deterministic parallelism advantageous for the repeatable, invariant solution path and results, whereas after development, during application deployment, you may prefer opportunistic parallelism for its performance.

To maintain determinism (that is, an implicit contract of repeatable, invariant search path and results), ILOG CPLEX invokes deterministic parallelism by default for MIP optimization, but when the *global default thread count* parameter (Threads, CPX_PARAM_THREADS) is set to a value strictly greater than one, opportunistic parallel MIP will be invoked. This higher setting implicitly specifies to ILOG CPLEX that the user is willing to accept opportunistic behavior.

In addition to the threads parameter, you can use the *parallel mode switch* (ParallelMode, CPX_PARAM_PARALLELMODE) to control the invocation of opportunistic algorithms. With its default setting of 0 (zero), only deterministic algorithms are used, unless the threads parameter is changed to a value strictly greater than one.

To force ILOG CPLEX to use deterministic algorithms in all cases, set the parallel mode parameter to 1 (one).

To allow ILOG CPLEX to use opportunistic algorithms in all situations, set the parallel mode parameter to -1 (minus one).

The presence of a time limit, set by the *optimizer time limit* parameter (TiLim, CPX_PARAM_TILIM) for example, poses problems for reproducibility in any algorithm that

is otherwise deterministic, even in sequential rather than parallel mode. Subtle variations in runtime on computer architectures can lead to variation in the measurement of time. (Other limits, such as node limits, are not subject to this variability.)

Because time limits are so important in many applications, ILOG CPLEX will still attempt to use deterministic mode without regard to whether a time limit is in effect. While variability will still be much lower than with the opportunistic setting, the user is advised that complete determinism within time limits can not be assured.

Using parallel optimizers in the Interactive Optimizer

For mixed integer programming (MIP), this policy means the following sequence of commands invokes parallel deterministic MIP optimization:

1. Start the parallel CPLEX Interactive Optimizer with the command `plex` at the operating system prompt.
2. Enter your problem object and populate it with data as usual.
3. Call the parallel optimizer with the `mipopt` command. The optimizer will use the maximum number of available threads in deterministic mode.

In order to invoke parallel optimization of *continuous* models in the Interactive Optimizer, you must abandon determinism. You can do so explicitly with the parallel mode parameter, or you can do so implicitly with the threads parameter.

To abandon determinism explicitly by means of the parallel mode parameter, do this:

1. Start the parallel CPLEX Interactive Optimizer with the command `plex` at the operating system prompt.
2. Enter your problem object and populate it with data as usual.
3. Enter the command: `set parallel -1`. The optimizer will use the maximum number of available threads in opportunistic mode.
4. Optimize.

To abandon determinism implicitly by means of the thread parameter, do this:

1. Start the parallel CPLEX Interactive Optimizer with the command `plex` at the operating system prompt.
2. Enter your problem object and populate it with data as usual.
3. Enter the command: `set threads n` where `n` is a number strictly greater than 1 (one). The optimizer will use the specified number of threads in opportunistic mode.
4. 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* and *Instantiate the problem as an object* for details.
2. Enter and populate your problem object in the usual way, as in *Put data in the problem object*.
3. If you want to use an opportunistic parallel algorithm, set either the *parallel mode switch* (ParallelMode, CPX_PARAM_PARALLELMODE) or the *global default thread count* (Threads, CPX_PARAM_THREADS) parameter in your application. This step is mandatory if you want to use the parallel barrier optimizer or the concurrent optimizer. If you want to use the parallel MIP optimizer in deterministic mode with fewer than the maximum number of available threads, set both the parallel mode and the threads parameter in your application. See *Determinism of results* for background about this step.
4. Call the parallel optimizer with the appropriate method or routine from *Parallel optimizer methods and routines of the Component Libraries*.

Parallel optimizer methods and routines of the 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*, and explored further in *Solving problems with a quadratic objective (QP)* and in *Solving problems with quadratic constraints (QCP)*.) Consequently, the parallel barrier optimizer will be the best continuous choice on a parallel computer more frequently than on a single-processor. For that reason, you should be careful not to apply performance data or experience based on serial optimizers when you are choosing which optimizer to use on a parallel platform.

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

Describes the parallel MIP optimizer.

In this section

Introducing parallel MIP optimization

Describes parallel MIP optimization in general terms.

Root relaxation and parallel MIP processing

Describes special considerations about the root relaxation in parallel MIP optimization.

Memory considerations and the parallel MIP optimizer

Describes considerations about memory consumption in parallel MIP optimization.

Output from the parallel MIP optimizer

Shows a sample log from parallel MIP optimization.

Introducing parallel MIP optimization

By default, ILOG CPLEX uses the *deterministic* parallel MIP optimizer to solve a mixed integer programming problem. In doing so, it exploits parallel computations while it solves nodes of the MIP branch & cut tree. It also executes strong branching computations in parallel.

Besides these applications of parallel processes, there may be additional possibilities for parallel computation at the root node. However, these additional possibilities are not deterministic; they are known as *opportunistic*. In order to exploit such opportunities, your application must allow ILOG CPLEX to run opportunistically in parallel. To do so, either you can explicitly set the *parallel mode switch* (`ParallelMode`, `CPX_PARAM_PARALLELMODE`) to the value -1 (minus one), or you can implicitly allow opportunistic parallel optimization by setting the *global default thread count* (`Threads`, `CPX_PARAM_THREADS`) parameter to a value strictly greater than one.

As explained in *Determinism of results*, you may find it advantageous to develop your application in deterministic parallel mode, where you can rely on the invariance and repeatability of the search path and results to evaluate the correctness of your model and solutions. After you are convinced of correctness of your model, there are two different approaches you can take in deployment of your application. If performance is critical, consider deploying in opportunistic parallel mode. While faster performance in opportunistic mode cannot be guaranteed, it does generally out-perform deterministic mode. On the other hand, if performance is not critical, you may prefer to deploy in deterministic mode to retain the possibility of reproducing any problems that your end-user may encounter during deployment. In summary, you need to evaluate for your model and application which mode is more appropriate.

Root relaxation and parallel MIP 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 (one), the concurrent optimizer is invoked by default. This provides a form of parallelism that assigns 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 *MIP starting algorithm* parameter:

- ◆ `RootAlg` in Concert Technology;
- ◆ `CPX_PARAM_STARTALG` in the Callable Library;
- ◆ `set mip strategy startalgorithm` in the Interactive Optimizer.

In such a case, the parallel threads will all be applied to the barrier algorithm.

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*) from the sequential MIP optimizer. The following paragraphs explain those differences.

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 = 0.00 sec.
Clique table members: 2.
MIP emphasis: balance optimality and feasibility.
Root relaxation solution time = 0.00 sec.

```

	Nodes				Cuts/			
	Node	Left	Objective	IInf	Best Integer	Best Node	ItCnt	Gap
*	0+	0			97863.0000	3192.0420	60	96.74%
*	0+	0			4505.0000	3192.0420	60	29.14%
	0	0	3383.7784	16	4505.0000	Cuts: 35	114	24.89%
	0	0	3489.7887	13	4505.0000	Cuts: 17	134	22.54%
	0	0	3531.4512	16	4505.0000	Cuts: 11	149	21.61%
*	0+	0			4501.0000	3531.4512	149	21.54%
	0	0	3552.4474	19	4501.0000	Cuts: 9	159	21.07%
	0	0	3588.7513	19	4501.0000	Cuts: 8	178	20.27%
	0	0	3597.3630	20	4501.0000	Cuts: 5	185	20.08%
	0	0	3598.6200	21	4501.0000	Cuts: 6	190	20.05%
	0	0	3601.2921	22	4501.0000	Cuts: 3	194	19.99%
*	0+	0			4457.0000	3601.2921	194	19.20%
	0	0	3602.5189	23	4457.0000	Cuts: 3	197	19.17%
	0	0	3633.5846	20	4457.0000	Cuts: 2	204	18.47%
	0	0	3638.8299	22	4457.0000	Covers: 1	206	18.36%
	0	0	3649.9331	22	4457.0000	Cuts: 3	212	18.11%
	0	0	3652.9758	28	4457.0000	Cuts: 5	218	18.04%
	0	0	3656.7563	24	4457.0000	Cuts: 3	227	17.95%
	0	0	3659.1865	29	4457.0000	Cuts: 4	232	17.90%
	0	0	3664.7373	26	4457.0000	Covers: 1	236	17.78%
	0	0	3676.2923	33	4457.0000	Covers: 5	244	17.52%
	0	0	3676.7448	31	4457.0000	Flowcuts: 2	250	17.51%
	0	0	3676.9154	33	4457.0000	Flowcuts: 2	254	17.50%
*	0+	0			3994.0000	3676.9154	254	7.94%
*	0+	0			3990.0000	3676.9154	254	7.85%
*	0+	0			3985.0000	3676.9154	254	7.73%
	0	2	3680.6918	22	3985.0000	3676.9154	254	7.73%
*	22	5	integral	0	3983.0000	3770.3216	358	5.34%

```

Root node processing (before b&c):
  Real time           =    1.39
Parallel b&c, 2 threads:
  Real time           =    0.24
  Sync time (average) =    0.08
  Wait time (average) =    0.03
  -----
Total (root+branch&cut) =    1.63 sec.

Cover cuts applied: 13
Flow cuts applied: 40
Gomory fractional cuts applied: 6
Solution pool: 8 solutions saved

MIP - Integer optimal solution: Objective = 3.9830000000e+03

Solution time =    1.63 sec.  Iterations = 426  Nodes = 40

```

The sample reports that the processors spent an average of 0.03 of a second of real time waiting for other processors to provide nodes to be processed. The Sync time is the average time a processor spends trying to synchronize by accessing globally shared data. Because only one processor at a time can access such data, the amount of time spent in synchronization is really crucial: any other processor that tries to access this region must wait, thus sitting idle, and this idle time is counted separately from the Wait time.

There is another difference in the way logging occurs in the parallel MIP optimizer. When this optimizer is called, it makes a number of copies of the problem. These copies are known as clones. The parallel MIP optimizer creates as many clones as there are threads available to it. When the optimizer exits, these clones and the memory they used are discarded.

If a log file is active when the clones are created, then ILOG CPLEX creates a clone log file for each clone. The clone log files are named `cloneK.log`, where K is the index of the clone, ranging from 0 (zero) to the number of threads minus one. Since the clones are created at each call to the parallel MIP optimizer and discarded when it exits, the clone logs are opened at each call and closed at each exit. (The clone log files are not removed when the clones themselves are discarded.)

The clone logs contain information normally recorded in the ordinary log file (by default, `cplex.log`) but inconvenient to send through the normal log channel. The information likely to be of most interest to you are special messages, such as error messages, that result from calls to the LP optimizers called for the subproblems.

Clock settings and time measurement

On most platforms, ILOG CPLEX offers two ways of measuring time: CPU time and wall-clock time. However, on Microsoft Windows platforms, only wall-clock time is available. By default, when ILOG CPLEX is running sequentially, CPU time is reported on platforms that support it. In contrast, when ILOG CPLEX is running parallel algorithms, wall-clock time is the default on all platforms.

On platforms that support both types of clock, you can choose the type of clock by setting the *clock type for computation time* parameter to a value different from the default 0 (zero).

- ◆ In Concert Technology, use the method:
 - `IloCplex::setParam(ClockType, i)` in the C++ API;
 - `IloCplex.setParam(ClockType, i)` in the Java API;
 - `Cplex.SetParam(ClockType, i)` in the .NET API.
- ◆ In the Callable Library, use the routine `CPXsetintparam(env, CPX_PARAM_CLOCKTYPE, i)`.
- ◆ In the Interactive Optimizer, use the command `set clocktype i`.

Replace the *i* with the value 1 (one) to specify CPU time or 2 to specify wall-clock time.

ILOG CPLEX Parameters Reference Manual

The behavior of CPLEX is controlled by a variety of parameters that are each accessible and settable by the user. This manual lists these parameters and explains their settings in the CPLEX Component Libraries and the Interactive Optimizer. It also explains how to read and write parameter settings of the C API to a file.

ILOG CPLEX Parameters Reference Manual

The behavior of CPLEX is controlled by a variety of parameters that are each accessible and settable by the user. This manual lists these parameters and explains their settings in the CPLEX Component Libraries and the Interactive Optimizer. It also explains how to read and write parameter settings of the C API to a file.

In this section

Accessing parameters

Identifies the accessors for parameters in the different APIs.

Parameter names

Explains the naming conventions of ILOG CPLEX parameters.

Correspondence of parameters

Associates parameters available in the Callable Library with those available in Concert Technology.

Saving parameter settings to a file

Describes PRM files.

Topical list of parameters

The following lists offer you access to the documentation of CPLEX parameters, organized by topics.

List of CPLEX parameters

Presents the entire list of parameters

Accessing parameters

The following methods set and access parameters for objects of the class `IloCplex` in C++ and Java or the class `Cplex` in the .NET API:

```
setParam  
getParam  
getMin  
getMax  
getDefault  
setDefault
```

The names of the corresponding accessors in the class `Cplex` in .NET follow the usual conventions of names and capitalization of languages in that framework. For example, the class `Cplex` and its method `Solve` are denoted `Cplex.Solve`.

C applications and applications written in other languages callable from C access and set parameters with the following routines:

<code>CPXgetdblparam</code>	Accesses a parameter of type double
<code>CPXsetdblparam</code>	Changes a parameter of type double
<code>CPXinfodblparam</code>	Gets the default value and range of a parameter of type double
<code>CPXgetintparam</code>	Accesses a parameter of type integer
<code>CPXsetintparam</code>	Changes a parameter of type integer
<code>CPXinfointparam</code>	Gets the default value and range of a parameter of type integer
<code>CPXgetstrparam</code>	Accesses a parameter of type string
<code>CPXsetstrparam</code>	Changes a parameter of type string
<code>CPXinfostrparam</code>	Gets the default value of a parameter of type string
<code>CPXsetdefaults</code>	Resets all parameters to their standard default values
<code>CPXgetparamname</code>	Accesses the name of a parameter
<code>CPXgetparamnum</code>	Access the identifying number assigned to a parameter
<code>CPXgetchparams</code>	Accesses all parameters not currently at their default value

Parameter names

In the parameter table, each parameter has a name (that is, a symbolic constant) to refer to it within a program.

- ◆ For the C API, these constants are capitalized and start with `CPX_PARAM_`; for example, `CPX_PARAM_ITLIM`. They are used as the second argument in all parameter routines (except `CPXsetdefaults` which does not require them).
- ◆ For C++ applications, the parameters are defined in nested enumeration types for Boolean, integer, floating-point, and string parameters. The `enum` names use mixed (lower and upper) case letters and must be prefixed with the class name `IloCplex::` for scope. For example, `IloCplex::ItLim` is the `IloCplex` equivalent of `CPX_PARAM_ITLIM`.
- ◆ For Java applications, the parameters are defined as final static objects in nested classes called `IloCplex.BooleanParam`, `IloCplex.IntParam`, `IloCplex.DoubleParam`, and `IloCplex.StringParam` for Boolean, integer, floating-point, and string parameters, respectively. The parameter object names use mixed (lower and upper) case letters and must be prefixed with the appropriate class for scope. For example, `IloCplex.IntParam.ItLim` is the object representing the parameter `CPX_PARAM_ITLIM`.
- ◆ For .NET applications, the parameters follow the usual conventions for capitalizing attributes and defining scope within a namespace.

An integer that serves as a reference number for each parameter is shown in the table. That integer reference number corresponds to the value that each symbolic constant represents, as found in the `plex.h` header file, but it is strongly recommended that the symbolic constants be used instead of their integer equivalents whenever possible, for the sake of portability to future versions of CPLEX.

Correspondence of parameters

Some parameters available for the C API are not supported as parameters for the object oriented APIs or have a slightly different name there. In particular:

- ◆ *epsilon used in linearization* (EpLin), the parameter specifying the tolerance to use in linearization in the object oriented APIs (C++, Java, .NET), is not applicable in the C API.
- ◆ *MIP callback switch between original model and reduced, presolved model* (CPX_PARAM_MIPCBREDLP), the parameter indicating whether to use the reduced or original model in MIP callbacks, has no equivalent in the object oriented APIs.
- ◆ Logging output is controlled by a parameter in the C API (CPX_PARAM_SCRIND), but when using the object oriented APIs, you control logging by configuring the output channel:
 - IloCplex::out in C++
For example, to turn off output to the screen, use `cplex.setOut(env.getNullStream())`.
 - IloCplex.output in Java
For example, to turn off output to the screen, use `cplex.setOut(null)`.
 - Cplex.Out in .NET
For example, to turn off output to the screen, use `Cplex.SetOut(Null)`.
- ◆ The parameter `IloCplex::RootAlg` in the C++ API corresponds to these parameters in the C API:
 - *MIP starting algorithm*: CPX_PARAM_STARTALG
 - *algorithm for continuous problems*: CPX_PARAM_LPMETHOD
 - *algorithm for continuous quadratic optimization*: CPX_PARAM_QPMETHOD
- ◆ The parameter `IloCplex::NodeAlg` in the C++ API corresponds to the parameter *MIP subproblem algorithm* CPX_PARAM_SUBALG in the C API.

Saving parameter settings to a file

It is possible to read and write a file of parameter settings with the C API. The file extension is `.prm`. The C routine `CPXreadcopyparam` reads parameter values from a file with the `.prm` extension. The routine `CPXwriteparam` writes a file of the current nondefault parameter settings to a file with the `.prm` extension. Here is the format of such a file:

```
CPLEX Parameter File Version number
parameter_name parameter_value
```

Tip: The heading with a version number in the first line of a PRM file is significant to ILOG CPLEX. An easy way to produce a correctly formatted PRM file with a proper heading is to have ILOG CPLEX write the file for you.

CPLEX reads the entire file before changing any of the parameter settings. After successfully reading a parameter file, the C API first sets all parameters to their default value. Then it applies the settings it read in the parameter file. No changes are made if the parameter file contains errors, such as missing or illegal values. There is no checking for duplicate entries in the file. In the case of duplicate entries, the last setting in the file is applied.

When you write a parameter file from the C API, only the non-default values are written to the file. String values may be double-quoted or not, but are always written with double quotation marks.

The comment character in a parameter file is `#`. After that character, CPLEX ignores the rest of the line.

The C API issues a warning if the version recorded in the parameter file does not match the version of the product. A warning is also issued if a nonintegral value is given for an integer-valued parameter.

Here is an example of a correct CPLEX parameter file:

```
CPLEX Parameter File Version 11.0
CPX_PARAM_EPPER          3.450000000000000e-06
CPX_PARAM_OBJULIM       1.23456789012345e+05
CPX_PARAM_PERIND        1
CPX_PARAM_SCRIND        1
CPX_PARAM_WORKDIR       "tmp"
```


Topical list of parameters

The following lists offer you access to the documentation of CPLEX parameters, organized by topics.

In this section

Simplex

Lists parameters of interest to users of the simplex optimizers.

Barrier

Lists parameters of interest to users of the barrier optimizer.

MIP

Lists topics of interest to users of the MIP optimizer.

MIP general

Lists parameters of general interest to users of the MIP optimizer.

MIP strategies

Lists parameters controlling MIP strategies.

MIP cuts

Lists parameters controlling cuts.

MIP tolerances

Lists parameters setting MIP tolerances.

MIP limits

Lists parameters setting MIP limits.

Solution polishing

Lists parameters controlling starting conditions for solution polishing

Solution pool

Lists parameters controlling the solution pool.

Network

Lists parameters of interest to users of the network flow optimizer.

Parallel optimization

Lists parameters controlling parallel optimization.

Sifting

Lists parameters of interest to users of the sifting optimizer.

Preprocessing: aggregator, presolver

Lists parameters related to preprocessing.

Tolerances

Lists parameters setting tolerances.

Limits

Lists parameters setting general limits.

Display and output

Lists parameters controlling screen displays, logs, and files.

Simplex

advanced start switch
lower objective value limit
upper objective value limit
dual simplex pricing algorithm
primal simplex pricing algorithm
simplex crash ordering
Markowitz tolerance
optimality tolerance
perturbation constant
simplex perturbation switch
simplex perturbation limit
relaxation for FeasOpt
feasibility tolerance
simplex maximum iteration limit
memory reduction switch
numerical precision emphasis
simplex pricing candidate list size
sifting subproblem algorithm
simplex iteration information display
simplex singularity repair limit

Barrier

advanced start switch
barrier algorithm
barrier starting point algorithm
barrier crossover algorithm
sifting subproblem algorithm
barrier ordering algorithm
barrier display information
barrier growth limit
barrier column nonzeros
barrier iteration limit
barrier maximum correction limit
barrier objective range
convergence tolerance for LP and QP problems
convergence tolerance for QC problems
memory reduction switch
numerical precision emphasis

MIP

The parameters controlling MIP behavior are accessible through the following topics:

- ◆ *MIP general*
- ◆ *MIP strategies*
- ◆ *MIP cuts*
- ◆ *MIP tolerances*
- ◆ *MIP limits*

MIP general

advanced start switch

MIP emphasis switch

MIP subproblem algorithm

MIP repeat presolve switch

relaxed LP presolve switch

indefinite MIQP switch

bound strengthening switch

memory reduction switch

numerical precision emphasis

MIP callback switch between original model and reduced, presolved model

MIP node log display information

MIP node log interval

node storage file switch

MIP strategies

MIP starting algorithm
MIP variable selection strategy
MIP strategy best bound interval
MIP branching direction
backtracking tolerance
MIP dive strategy
MIP heuristic frequency
local branching heuristic
MIP priority order switch
MIP priority order generation
MIP node selection strategy
node presolve switch
MIP probing level
RINS heuristic frequency
feasibility pump switch

MIP cuts

constraint aggregation limit for cut generation

row multiplier factor for cuts

MIP cliques switch

MIP covers switch

MIP disjunctive cuts switch

MIP flow cover cuts switch

MIP flow path cut switch

MIP Gomory fractional cuts switch

MIP GUB cuts switch

MIP implied bound cuts switch

MIP MIR (mixed integer rounding) cut switch

MIP zero-half cuts switch

pass limit for generating Gomory fractional cuts

candidate limit for generating Gomory fractional cuts

type of cut limit

number of cutting plane passes

MIP tolerances

backtracking tolerance

lower cutoff

upper cutoff

absolute objective difference cutoff

relative objective difference cutoff

absolute MIP gap tolerance

relative MIP gap tolerance

integrality tolerance

relaxation for FeasOpt

MIP limits

MIP integer solution limit
pass limit for generating Gomory fractional cuts
candidate limit for generating Gomory fractional cuts
constraint aggregation limit for cut generation
type of cut limit
row multiplier factor for cuts
number of cutting plane passes
MIP node limit
time before starting to polish a feasible solution
time spent probing
frequency to try to repair infeasible MIP start
MIP strong branching candidate list limit
MIP strong branching iterations limit
limit on nodes explored when a subMIP is being solved
tree memory limit

Solution polishing

absolute MIP gap before starting to polish a feasible solution

relative MIP gap before starting to polish a feasible solution

MIP integer solutions to find before starting to polish a feasible solution

nodes to process before starting to polish a feasible solution

time before starting to polish a feasible solution

Solution pool

solution pool intensity

solution pool replacement strategy

limit on number of solutions generated for solution pool

limit on number of solutions kept in solution pool

absolute gap for solution pool

relative gap for solution pool

Network

network optimality tolerance

network primal feasibility tolerance

simplex network extraction level

network simplex iteration limit

network simplex pricing algorithm

network logging display switch

Parallel optimization

parallel mode switch

global default thread count

Sifting

sifting subproblem algorithm

sifting information display

upper limit on sifting iterations

Preprocessing: aggregator, presolver

symmetry breaking
preprocessing aggregator fill
preprocessing aggregator application limit
bound strengthening switch
coefficient reduction setting
dependency switch
presolve dual setting
presolve switch
linear reduction switch
limit on the number of presolve passes made
node presolve switch
relaxed LP presolve switch
MIP repeat presolve switch
primal and dual reduction type

Tolerances

convergence tolerance for LP and QP problems

convergence tolerance for QC problems

backtracking tolerance

lower cutoff

upper cutoff

absolute MIP gap tolerance

absolute MIP gap before starting to polish a feasible solution

relative MIP gap tolerance

relative MIP gap before starting to polish a feasible solution

integrality tolerance

epsilon used in linearization

Markowitz tolerance

optimality tolerance

network optimality tolerance

feasibility tolerance

relaxation for FeasOpt

absolute objective difference cutoff

relative objective difference cutoff

perturbation constant

absolute gap for solution pool

relative gap for solution pool

Limits

memory available for working storage

global default thread count

optimizer time limit

variable (column) read limit

constraint (row) read limit

nonzero element read limit

QP Q-matrix nonzero read limit

Display and output

messages to screen switch

tuning information display

barrier display information

simplex iteration information display

sifting information display

MIP node log display information

MIP node log interval

network logging display switch

clock type for computation time

conflict information display

data consistency checking switch

precision of numerical output in MPS and REW file formats

directory for working files

write level for MST, SOL files

List of CPLEX parameters

Presents the entire list of parameters

In this section

advanced start switch

If set to 1 or 2, this parameter indicates that CPLEX should use advanced starting information when optimization is initiated.

constraint aggregation limit for cut generation

Limits the number of constraints that can be aggregated for generating flow cover and mixed integer rounding (MIR) cuts.

preprocessing aggregator fill

Limits variable substitutions by the aggregator.

preprocessing aggregator application limit

Invokes the aggregator to use substitution where possible to reduce the number of rows and columns before the problem is solved.

barrier algorithm

The default setting 0 uses the "infeasibility - estimate start" algorithm (setting 1) when solving subproblems in a MIP problem, and the standard barrier algorithm (setting 3) in other cases.

barrier column nonzeros

Used in the recognition of dense columns.

barrier crossover algorithm

Decides which, if any, crossover is performed at the end of a barrier optimization.

barrier display information

Sets the level of barrier progress information to be displayed.

convergence tolerance for LP and QP problems

Sets the tolerance on complementarity for convergence.

barrier growth limit

Used to detect unbounded optimal faces.

barrier iteration limit

Sets the number of barrier iterations before termination.

barrier maximum correction limit

Sets the maximum number of centering corrections done on each iteration.

barrier objective range

Sets the maximum absolute value of the objective function.

barrier ordering algorithm

Sets the algorithm to be used to permute the rows of the constraint matrix in order to reduce fill in the Cholesky factor.

convergence tolerance for QC problems

Sets the tolerance on complementarity for convergence in quadratically constrained problems (QCPs).

barrier starting point algorithm

Sets the algorithm to be used to compute the initial starting point for the barrier optimizer.

MIP strategy best bound interval

Sets the best bound interval for MIP strategy.

bound strengthening switch

Decides whether to apply bound strengthening in mixed integer programs (MIPs).

MIP branching direction

Decides which branch, the up or the down branch, should be taken first at each node.

backtracking tolerance

Controls how often backtracking is done during the branching process.

MIP cliques switch

Decides whether or not clique cuts should be generated for the problem.

clock type for computation time

Decides how computation times are measured for both reporting performance and terminating optimization when a time limit has been set.

coefficient reduction setting

Decides how coefficient reduction is used.

variable (column) read limit

Specifies a limit for the number of columns (variables) to read for an allocation of memory.

conflict information display

Decides how much information CPLEX reports when the conflict refiner is working.

MIP covers switch

Decides whether or not cover cuts should be generated for the problem.

simplex crash ordering

Decides how CPLEX orders variables relative to the objective function when selecting an initial basis.

lower cutoff

Sets lower cutoff tolerance.

number of cutting plane passes

Sets the upper limit on the number of cutting plane passes CPLEX performs when solving the root node of a MIP model.

row multiplier factor for cuts

Limits the number of cuts that can be added.

upper cutoff

Sets the upper cutoff tolerance.

data consistency checking switch

Decides whether data should be checked for consistency.

dependency switch

Decides whether to activate the dependency checker.

MIP disjunctive cuts switch

Decides whether or not disjunctive cuts should be generated for the problem.

MIP dive strategy

Controls the MIP dive strategy.

dual simplex pricing algorithm

Decides the type of pricing applied in the dual simplex algorithm.

type of cut limit

Sets a limit for each type of cut.

absolute MIP gap tolerance

Sets an absolute tolerance on the gap between the best integer objective and the objective of the best node remaining.

relative MIP gap tolerance

Sets a relative tolerance on the gap between the best integer objective and the objective of the best node remaining.

integrality tolerance

Specifies the amount by which an integer variable can be different from an integer and still be considered feasible.

epsilon used in linearization

Sets the epsilon (degree of tolerance) used in linearization in the object-oriented APIs.

Markowitz tolerance

Influences pivot selection during basis factoring.

optimality tolerance

Influences the reduced-cost tolerance for optimality.

perturbation constant

Sets the amount by which CPLEX perturbs the upper and lower bounds or objective coefficients on the variables when a problem is perturbed in the simplex algorithm.

relaxation for FeasOpt

Controls the amount of relaxation for the routine `CPXfeasopt` in the C API or for the method `feasOpt` in the object-oriented APIs.

feasibility tolerance

Specifies the feasibility tolerance, that is, the degree to which the basic variables of a model may violate their bounds.

mode of FeasOpt

Decides how FeasOpt measures the relaxation when finding a minimal relaxation in an infeasible model.

MIP flow cover cuts switch

Decides whether or not to generate flow cover cuts for the problem.

MIP flow path cut switch

Decides whether or not flow path cuts should be generated for the problem.

feasibility pump switch

Turns on or off the feasibility pump heuristic for mixed integer programming (MIP) models.

candidate limit for generating Gomory fractional cuts

Limits the number of candidate variables for generating Gomory fractional cuts.

MIP Gomory fractional cuts switch

Decides whether or not Gomory fractional cuts should be generated for the problem.

pass limit for generating Gomory fractional cuts

Limits the number of passes for generating Gomory fractional cuts.

MIP GUB cuts switch

Decides whether or not to generate GUB cuts for the problem.

MIP heuristic frequency

Decides how often to apply the periodic heuristic.

MIP implied bound cuts switch

Decides whether or not to generate implied bound cuts for the problem.

MIP integer solution limit

Sets the number of MIP solutions to be found before stopping.

simplex maximum iteration limit

Sets the maximum number of simplex iterations to be performed before the algorithm terminates without reaching optimality.

local branching heuristic

Controls whether CPLEX applies a local branching heuristic to try to improve new incumbents found during a MIP search.

memory reduction switch

Directs CPLEX that it should conserve memory where possible.

MIP callback switch between original model and reduced, presolved model

Controls whether your callback accesses node information of the original model (off) or node information of the reduced, presolved model (on, default).

MIP node log display information

Decides what CPLEX reports to the screen during mixed integer optimization (MIP).

MIP emphasis switch

Controls trade-offs between speed, feasibility, optimality, and moving bounds in MIP.

MIP node log interval

Controls the frequency of node logging when the MIP display parameter (CPX_PARAM_MIPDISPLAY, `MIPDisplay`) is set higher than 1 (one).

MIP priority order switch

Decides whether to use the priority order, if one exists, for the next mixed integer optimization.

MIP priority order generation

Selects the type of generic priority order to generate when no priority order is present.

MIP dynamic search switch

Sets the search strategy for a mixed integer program (MIP).

MIQCP strategy switch

Sets the strategy that CPLEX uses to solve a quadratically constrained mixed integer program (MIQCP).

MIP MIR (mixed integer rounding) cut switch

Decides whether or not to generate MIR cuts (mixed integer rounding cuts) for the problem.

precision of numerical output in MPS and REW file formats

Decides the precision of numerical output in the MPS and REW file formats.

network logging display switch

Decides what CPLEX reports to the screen during network optimization.

network optimality tolerance

Specifies the optimality tolerance for network optimization.

network primal feasibility tolerance

Specifies feasibility tolerance for network primal optimization. The feasibility tolerance specifies the degree to which the flow value of a model may violate its bounds.

simplex network extraction level

Establishes the level of network extraction for network simplex optimization.

network simplex iteration limit

Sets the maximum number of iterations to be performed before the algorithm terminates without reaching optimality.

network simplex pricing algorithm

Specifies the pricing algorithm for network simplex optimization.

MIP subproblem algorithm

Decides which continuous optimizer will be used to solve the subproblems in a MIP, after the initial relaxation.

node storage file switch

Used when working memory (CPX_PARAM_WORKMEM, WorkMem) has been exceeded by the size of the tree.

MIP node limit

Sets the maximum number of nodes solved before the algorithm terminates without reaching optimality.

MIP node selection strategy

Used to set the rule for selecting the next node to process when backtracking.

numerical precision emphasis

Emphasizes precision in numerically unstable or difficult problems.

nonzero element read limit

Specifies a limit for the number of nonzero elements to read for an allocation of memory.

absolute objective difference cutoff

Used to update the cutoff each time a mixed integer solution is found.

lower objective value limit

Sets a lower limit on the value of the objective function in the simplex algorithms.

upper objective value limit

Sets an upper limit on the value of the objective function in the simplex algorithms.

parallel mode switch

Sets the parallel optimization mode. Possible modes are automatic, deterministic, and opportunistic.

simplex perturbation switch

Decides whether to perturb problems.

simplex perturbation limit

Sets the number of degenerate iterations before perturbation is performed.

absolute MIP gap before starting to polish a feasible solution

Sets an absolute MIP gap after which CPLEX starts to polish a feasible solution

relative MIP gap before starting to polish a feasible solution

Sets a relative MIP gap after which CPLEX starts to polish a feasible solution

MIP integer solutions to find before starting to polish a feasible solution

Sets the number of integer solutions to find after which CPLEX starts to polish a feasible solution

nodes to process before starting to polish a feasible solution

Sets the number of nodes to process after which CPLEX starts to polish a feasible solution

time before starting to polish a feasible solution

Sets the amount of time in seconds to spend during a normal mixed integer optimization after which CPLEX starts to polish a feasible solution

time spent polishing a solution (deprecated)

Deprecated parameter

limit on number of solutions generated for solution pool

Limits the number of mixed integer programming (MIP) solutions generated for the solution pool during each call to the populate procedure.

primal simplex pricing algorithm

Sets the primal simplex pricing algorithm.

resolve dual setting

Decides whether CPLEX presolve should pass the primal or dual linear programming problem to the linear programming optimization algorithm.

presolve switch

Decides whether CPLEX applies presolve during preprocessing.

linear reduction switch

Decides whether linear or full reductions occur during preprocessing.

limit on the number of presolve passes made

Limits the number of presolve passes that CPLEX makes during preprocessing. When this parameter is set to a nonzero value, invokes CPLEX presolve to simplify and reduce problems.

node presolve switch

Decides whether node presolve should be performed at the nodes of a mixed integer programming (MIP) solution.

simplex pricing candidate list size

Sets the maximum number of variables kept in the list of pricing candidates for the simplex algorithms.

MIP probing level

Sets the amount of probing on variables to be performed before MIP branching.

time spent probing

Limits the amount of time in seconds spent probing.

indefinite MIQP switch

Decides whether CPLEX will attempt to reformulate a MIQP or MIQCP model that contains only binary variables.

QP Q-matrix nonzero read limit

Specifies a limit for the number of nonzero elements to read for an allocation of memory in a model with a quadratic matrix.

primal and dual reduction type

Decides whether primal reductions, dual reductions, both, or neither are performed during preprocessing.

simplex refactoring frequency

Sets the number of iterations between refactoring of the basis matrix.

relaxed LP presolve switch

Decides whether LP presolve is applied to the root relaxation in a mixed integer program (MIP).

relative objective difference cutoff

Used to update the cutoff each time a mixed integer solution is found.

frequency to try to repair infeasible MIP start

Limits the attempts to repair an infeasible MIP start.

MIP repeat presolve switch

Decides whether to re-apply presolve, with or without cuts, to a MIP model after processing at the root is otherwise complete.

RINS heuristic frequency

Decides how often to apply the relaxation induced neighborhood search (RINS) heuristic.

algorithm for continuous problems

Controls which algorithm is used to solve continuous models or to solve the root relaxation of a MIP.

algorithm for continuous quadratic optimization

Sets which algorithm to use when the C routine `CPXqpopt` (or the command `optimize` in the Interactive Optimizer) is invoked.

MIP starting algorithm

Sets which continuous optimizer will be used to solve the initial relaxation of a MIP.

constraint (row) read limit

Specifies a limit for the number of rows (constraints) to read for an allocation of memory.

scale parameter

Decides how to scale the problem matrix.

messages to screen switch

Decides whether or not results are displayed on screen in an application of the C API.

sifting subproblem algorithm

Sets the algorithm to be used for solving sifting subproblems.

sifting information display

Sets the amount of information to display about the progress of sifting.

upper limit on sifting iterations

Sets the maximum number of sifting iterations that may be performed if convergence to optimality has not been reached.

simplex iteration information display

Sets how often CPLEX reports about iterations during simplex optimization.

simplex singularity repair limit

Restricts the number of times CPLEX attempts to repair the basis when singularities are encountered during the simplex algorithm.

absolute gap for solution pool

Sets an absolute tolerance on the objective value for the solutions in the solution pool.

limit on number of solutions kept in solution pool

Limits the number of solutions kept in the solution pool

relative gap for solution pool

Sets a relative tolerance on the objective value for the solutions in the solution pool.

solution pool intensity

Controls the trade-off between the number of solutions generated for the solution pool and the amount of time or memory consumed.

solution pool replacement strategy

Designates the strategy for replacing a solution in the solution pool when the solution pool has reached its capacity.

MIP strong branching candidate list limit

Controls the length of the candidate list when CPLEX uses variable selection as the setting for strong branching.

MIP strong branching iterations limit

Controls the number of simplex iterations performed on each variable in the candidate list when CPLEX uses variable selection as the setting for strong branching.

limit on nodes explored when a subMIP is being solved

Restricts the number of nodes explored when CPLEX is solving a subMIP.

symmetry breaking

Decides whether symmetry breaking reductions will be automatically executed, during the preprocessing phase, in a MIP model.

global default thread count

Sets the default number of parallel threads that will be invoked by any CPLEX parallel optimizer.

optimizer time limit

Sets the maximum time, in seconds, for a call to an optimizer. This time limit applies also to the conflict refiner.

tree memory limit

Sets an absolute upper limit on the size (in megabytes, uncompressed) of the branch & cut tree.

tuning information display

Specifies the level of information reported by the tuning tool as it works.

tuning measure

Controls the measure for evaluating progress when a suite of models is being tuned.

tuning repeater

Specifies the number of times tuning is to be repeated on reordered versions of a given problem.

tuning time limit

Sets a time limit per model and per test set (that is, suite of models) applicable in tuning.

MIP variable selection strategy

Sets the rule for selecting the branching variable at the node which has been selected for branching.

directory for working files

Specifies the name of an existing directory into which CPLEX may store temporary working files.

memory available for working storage

Specifies an upper limit on the amount of central memory, in megabytes, that CPLEX is permitted to use for working memory.

write level for MST, SOL files

Sets a level of detail for CPLEX to write a file in MST or SOL format.

MIP zero-half cuts switch

Decides whether or not to generate zero-half cuts for the problem.

advanced start switch

Purpose

Advanced start switch

Syntax

C Name	CPX_PARAM_ADVIND (int)
C++ Name	AdvInd (int)
Java Name	AdvInd (int)
.NET Name	AdvInd (int)
OPL Name	advind
InteractiveOptimizer	advance
Identifier	1001

Description

If set to 1 or 2, this parameter indicates that CPLEX should use advanced starting information when optimization is initiated.

For MIP models, setting 1 (one) will cause CPLEX to continue with a partially explored MIP tree if one is available. If tree exploration has not yet begun, setting 1 (one) specifies that CPLEX should use a loaded MIP start, if available. Setting 2 retains the current incumbent (if there is one), re-applies presolve, and starts a new search from a new root.

Setting 2 is useful for continuous models. Consequently, it can be particularly useful for solving fixed MIP models, where a start vector but no corresponding basis is available.

For continuous models solved with simplex, setting 1 (one) will use the currently loaded basis. If a basis is available only for the original, unresolved model, or if CPLEX has a start vector rather than a simplex basis, then the simplex algorithm will proceed on the unresolved model. With setting 2, CPLEX will first perform presolve on the model and on the basis or start vector, and then proceed with optimization on the presolved problem.

For continuous models solved with the barrier algorithm, settings 1 or 2 will continue simplex optimization from the last available barrier iterate.

Values

Value	Meaning
--------------	----------------

0	Do not use advanced start information
1	Use an advanced basis supplied by the user; default
2	Crush an advanced basis or starting vector supplied by the user

constraint aggregation limit for cut generation

Purpose

Constraint aggregation limit for cut generation

Syntax

C Name	CPX_PARAM_AGGCUTLIM (int)
C++ Name	AggCutLim (int)
Java Name	AggCutLim (int)
.NET Name	AggCutLim (int)
OPL Name	aggcutlim
Interactive Optimizer	mip limits aggforcut
Identifier	2054

Description

Limits the number of constraints that can be aggregated for generating flow cover and mixed integer rounding (MIR) cuts.

Values

Any nonnegative integer; **default:** 3

preprocessing aggregator fill

Purpose

Preprocessing aggregator fill

Syntax

C Name	CPX_PARAM_AGGFILL (int)
C++ Name	AggFill (int)
Java Name	AggFill (int)
.NET Name	AggFill (int)
OPL Name	aggfill
Interactive Optimizer	preprocessing fill
Identifier	1002

Description

Limits variable substitutions by the aggregator. If the net result of a single substitution is more nonzeros than this value, the substitution is not made.

Values

Any nonnegative integer; **default:** 10

preprocessing aggregator application limit

Purpose

Preprocessing aggregator application limit

Syntax

C Name	CPX_PARAM_AGGIND (int)
C++ Name	AggInd (int)
Java Name	AggInd (int)
.NET Name	AggInd (int)
OPL Name	aggind
Interactive Optimizer	preprocessing aggregator
Identifier	1003

Description

Invokes the aggregator to use substitution where possible to reduce the number of rows and columns before the problem is solved. If set to a positive value, the aggregator is applied the specified number of times or until no more reductions are possible.

Values

Value	Meaning
-1	Automatic (1 for LP, infinite for MIP) default
0	Do not use any aggregator
Any positive integer	Number of times to apply aggregator

barrier algorithm

Purpose

Barrier algorithm

Syntax

C Name	CPX_PARAM_BARALG (int)
C++ Name	BarAlg (int)
Java Name	BarAlg (int)
.NET Name	BarAlg (int)
OPL Name	baralg
Interactive Optimizer	barrier algorithm
Identifier	3007

Description

The default setting 0 uses the "infeasibility - estimate start" algorithm (setting 1) when solving subproblems in a MIP problem, and the standard barrier algorithm (setting 3) in other cases. The standard barrier algorithm is almost always fastest. However, on problems that are primal or dual infeasible (common for MIP subproblems), the standard algorithm may not work as well as the alternatives. The two alternative algorithms (settings 1 and 2) may eliminate numerical difficulties related to infeasibility, but are generally slower.

Values

Value	Meaning
0	Default setting
1	Infeasibility-estimate start
2	Infeasibility-constant start
3	Standard barrier

barrier column nonzeros

Purpose

Barrier column nonzeros

Syntax

C Name	CPX_PARAM_BARCOLNZ (int)
C++ Name	BarColNz (int)
Java Name	BarColNz (int)
.NET Name	BarColNz (int)
OPL Name	barcolnz
Interactive Optimizer	barrier colnonzeros
Identifier	3009

Description

Used in the recognition of dense columns. If columns in the presolved and aggregated problem exist with more entries than this value, such columns are considered dense and are treated specially by the CPLEX Barrier Optimizer to reduce their effect.

Value	Meaning
0	Dynamically calculated; default
Any positive integer	Number of nonzero entries that make a column dense

barrier crossover algorithm

Purpose

Barrier crossover algorithm

Syntax

C Name	CPX_PARAM_BARCROSSALG (int)
C++ Name	BarCrossAlg (int)
Java Name	BarCrossAlg (int)
.NET Name	BarCrossAlg (int)
OPL Name	barcrossalg
Interactive Optimizer	barrier crossover
Identifier	3018

Description

Decides which, if any, crossover is performed at the end of a barrier optimization. This parameter also applies when CPLEX uses the Barrier Optimizer to solve an LP or QP problem, or when it is used to solve the continuous relaxation of an MILP or MIQP at a node in a MIP.

Value	Meaning
-------	---------

-1	No crossover
0	Automatic: let CPLEX choose; default
1	Primal crossover
2	Dual crossover

barrier display information

Purpose

Barrier display information

Syntax

C Name	CPX_PARAM_BARDISPLAY (int)
C++ Name	BarDisplay (int)
Java Name	BarDisplay (int)
.NET Name	BarDisplay (int)
OPL Name	bardisplay
Interactive Optimizer	barrier display
Identifier	3010

Description

Sets the level of barrier progress information to be displayed.

Value	Meaning
-------	---------

0	No progress information
1	Normal setup and iteration information; default
2	Diagnostic information

convergence tolerance for LP and QP problems

Purpose

Convergence tolerance for LP and QP problems

Syntax

C Name	CPX_PARAM_BAREPCOMP (double)
C++ Name	BarEpComp (double)
Java Name	BarEpComp (double)
.NET Name	BarEpComp (double)
OPL Name	barepcomp
Interactive Optimizer	barrier convergetol
Identifier	3002

Description

Sets the tolerance on complementarity for convergence. The barrier algorithm terminates with an optimal solution if the relative complementarity is smaller than this value.

Changing this tolerance to a smaller value may result in greater numerical precision of the solution, but also increases the chance of failure to converge in the algorithm and consequently may result in no solution at all. Therefore, caution is advised in deviating from the default setting.

Values

Any positive number greater than or equal to 1e-12; **default:** 1e-8.

See also

For problems with quadratic constraints (QCP), see *convergence tolerance for QC problems*

barrier growth limit

Purpose

Barrier growth limit

Syntax

C Name	CPX_PARAM_BARGROWTH (double)
C++ Name	BarGrowth (double)
Java Name	BarGrowth (double)
.NET Name	BarGrowth (double)
OPL Name	bargrowth
Interactive Optimizer	barrier limits growth
Identifier	3003

Description

Used to detect unbounded optimal faces. At higher values, the barrier algorithm is less likely to conclude that the problem has an unbounded optimal face, but more likely to have numerical difficulties if the problem has an unbounded face.

Values

1.0 or greater; **default:** 1e12.

barrier iteration limit

Purpose

Barrier iteration limit

Syntax

C Name	CPX_PARAM_BARITLIM (int)
C++ Name	BarItLim (int)
Java Name	BarItLim (int)
.NET Name	BarItLim (int)
OPL Name	baritlim
Interactive Optimizer	barrier limits iterations
Identifier	3012

Description

Sets the number of barrier iterations before termination. When this parameter is set to 0 (zero), no barrier iterations occur, but problem setup occurs and information about the setup is displayed (such as Cholesky factor statistics).

Values

Value	Meaning
0	No barrier iterations
2100000000	default
Any positive integer	Number of barrier iterations before termination

barrier maximum correction limit

Purpose

Barrier maximum correction limit

Syntax

C Name	CPX_PARAM_BARMAXCOR (int)
C++ Name	BarMaxCor (int)
Java Name	BarMaxCor (int)
.NET Name	BarMaxCor (int)
OPL Name	barmaxcor
Interactive Optimizer	barrier limits corrections
Identifier	3013

Description

Sets the maximum number of centering corrections done on each iteration. An explicit value greater than 0 (zero) may improve the numerical performance of the algorithm at the expense of computation time.

Values

Value	Meaning
-1	Automatic; let CPLEX choose; default
0	None
Any positive integer	Maximum number of centering corrections per iteration

barrier objective range

Purpose

Barrier objective range

Syntax

C Name	CPX_PARAM_BAROBJRNG (double)
C++ Name	BarObjRng (double)
Java Name	BarObjRng (double)
.NET Name	BarObjRng (double)
OPL Name	barobjrng
Interactive Optimizer	barrier limits objrange
Identifier	3004

Description

Sets the maximum absolute value of the objective function. The barrier algorithm looks at this limit to detect unbounded problems.

Values

Any nonnegative number; **default:** 1e20

barrier ordering algorithm

Purpose

Barrier ordering algorithm

Syntax

C Name	CPX_PARAM_BARORDER (int)
C++ Name	BarOrder (int)
Java Name	BarOrder (int)
.NET Name	BarOrder (int)
OPL Name	barorder
Interactive Optimizer	barrier ordering
Identifier	3014

Description

Sets the algorithm to be used to permute the rows of the constraint matrix in order to reduce fill in the Cholesky factor.

Values

Value	Meaning
0	Automatic: let CPLEX choose; default
1	Approximate minimum degree (AMD)
2	Approximate minimum fill (AMF)
3	Nested dissection (ND)

convergence tolerance for QC problems

Purpose

Convergence tolerance for quadratically constrained problems

Syntax

C Name	CPX_PARAM_BARQCPEPCOMP (double)
C++ Name	BarQCPEpComp (double)
Java Name	BarQCPEpComp (double)
.NET Name	BarQCPEpComp (double)
OPL Name	barqcpepcomp
Interactive Optimizer	barrier qcpcvergetol
Identifier	3020

Description

Sets the tolerance on complementarity for convergence in quadratically constrained problems (QCPs). The barrier algorithm terminates with an optimal solution if the relative complementarity is smaller than this value.

Changing this tolerance to a smaller value may result in greater numerical precision of the solution, but also increases the chance of a convergence failure in the algorithm and consequently may result in no solution at all. Therefore, caution is advised in deviating from the default setting.

Values

Any positive number greater than or equal to 1e-12; **default:** 1e-7.

For LPs and for QPs (that is, when all the constraints are linear) see *convergence tolerance for LP and QP problems* CPX_PARAM_BAREPCOMP, BarEpComp.

barrier starting point algorithm

Purpose

Barrier starting point algorithm

Syntax

C Name	CPX_PARAM_BARSTARTALG (int)
C++ Name	BarStartAlg (int)
Java Name	BarStartAlg (int)
.NET Name	BarStartAlg (int)
OPL Name	barstartalg
Interactive Optimizer	barrier startalg
Identifier	3017

Description

Sets the algorithm to be used to compute the initial starting point for the barrier optimizer.

Value	Meaning
-------	---------

1	Dual is 0 (zero); default
2	Estimate dual
3	Average of primal estimate, dual 0 (zero)
4	Average of primal estimate, estimate dual

MIP strategy best bound interval

Purpose

MIP strategy best bound interval

Syntax

C Name	CPX_PARAM_BBINTERVAL (int)
C++ Name	BBInterval (int)
Java Name	BBInterval (int)
.NET Name	BBInterval (int)
OPL Name	bbinterval
Interactive Optimizer	mip strategy bbinterval
Identifier	2039

Description

Sets the best bound interval for MIP strategy.

When you set this parameter to best estimate node selection, the best bound interval is the interval at which the best bound node, instead of the best estimate node, is selected from the tree. A best bound interval of 0 (zero) means “never select the best bound node.” A best bound interval of 1 (one) means “always select the best bound node,” and is thus equivalent to `nodeselect 1 (one)`.

Higher values of this parameter mean that the best bound node will be selected less frequently; experience has shown it to be beneficial to select the best bound node occasionally, and therefore the default value of this parameter is 7.

Values

Value	Meaning
0	Never select best bound node; always select best estimate
1	Always select best bound node
7	Select best bound node occasionally; default
Any positive integer	Select best bound node less frequently than best estimate node

See also

MIP node selection strategy

bound strengthening switch

Purpose

Bound strengthening switch

Syntax

C Name	CPX_PARAM_BNDSTRENIND (int)
C++ Name	BndStrenInd (int)
Java Name	BndStrenInd (int)
.NET Name	BndStrenInd (int)
OPL Name	bndstrenind
Interactive Optimizer	preprocessing boundstrength
Identifier	2029

Description

Decides whether to apply bound strengthening in mixed integer programs (MIPs). Bound strengthening tightens the bounds on variables, perhaps to the point where the variable can be fixed and thus removed from consideration during branch & cut.

Value Meaning

-1	Automatic: let CPLEX choose; default
0	Do not apply bound strengthening
1	Apply bound strengthening

MIP branching direction

Purpose

MIP branching direction

Syntax

C Name	CPX_PARAM_BRDIR (int)
C++ Name	BrDir (int)
Java Name	BrDir (int)
.NET Name	BrDir (int)
OPL Name	brdir
Interactive Optimizer	mip strategy branch
Identifier	2001

Description

Decides which branch, the up or the down branch, should be taken first at each node.

Value	Symbol	Meaning
-1	CPX_BRDIR_DOWN	Down branch selected first
0	CPX_BRDIR_AUTO	Automatic: let CPLEX choose; default
1	CPX_BRDIR_UP	Up branch selected first

backtracking tolerance

Purpose

Backtracking tolerance

Syntax

C Name	CPX_PARAM_BBTOL (double)
C++ Name	BtTol (double)
Java Name	BtTol (double)
.NET Name	BtTol (double)
OPL Name	bbtol
Interactive Optimizer	mip strategy backtrack
Identifier	2002

Description

Controls how often backtracking is done during the branching process. The decision when to backtrack depends on three values that change during the course of the optimization:

- ◆ the objective function value of the best integer feasible solution (*incumbent*)
- ◆ the best remaining objective function value of any unexplored node (*best node*)
- ◆ the objective function value of the most recently solved node (*current objective*).

If a cutoff tolerance (*upper cutoff* or *lower cutoff*) has been set by the user, then that value is used as the incumbent until an integer feasible solution is found.

The *target gap* is defined to be the absolute value of the difference between the incumbent and the best node, multiplied by this backtracking parameter. CPLEX does not backtrack until the absolute value of the difference between the objective of the current node and the best node is at least as large as the target gap.

Low values of this backtracking parameter thus tend to increase the amount of backtracking, which makes the search process more of a pure best-bound search. Higher parameter values tend to decrease backtracking, making the search more of a pure depth-first search.

The backtracking value has effect only after an integer feasible solution is found or when a cutoff has been specified. Note that this backtracking value merely permits backtracking but

does not force it; CPLEX may choose to continue searching a limb of the tree if that limb seems a promising candidate for finding an integer feasible solution.

Values

Any number from 0.0 to 1.0; **default:** 0.9999

See also

upper cutoff, lower cutoff

MIP cliques switch

Purpose

MIP cliques switch

Syntax

C Name	CPX_PARAM_CLIQUES (int)
C++ Name	Cliques (int)
Java Name	Cliques (int)
.NET Name	Cliques (int)
OPL Name	cliques
Interactive Optimizer	mip cuts cliques
Identifier	2003

Description

Decides whether or not clique cuts should be generated for the problem. Setting the value to 0 (zero), the default, indicates that the attempt to generate cliques should continue only if it seems to be helping.

Value Meaning

-1	Do not generate clique cuts
0	Automatic: let CPLEX choose; default
1	Generate clique cuts moderately
2	Generate clique cuts aggressively
3	Generate clique cuts very aggressively

clock type for computation time

Purpose

Clock type for computation time

Syntax

C Name	CPX_PARAM_CLOCKTYPE (int)
C++ Name	ClockType (int)
Java Name	ClockType (int)
.NET Name	ClockType (int)
OPL Name	clocktype
Interactive Optimizer	clocktype
Identifier	1006

Description

Decides how computation times are measured for both reporting performance and terminating optimization when a time limit has been set. Small variations in measured time on identical runs may be expected on any computer system with any setting of this parameter.

The default setting 0 (zero) allows ILOG CPLEX to choose wall clock time when other parameters invoke parallel optimization and to choose CPU time when other parameters enforce sequential (not parallel) optimization.

Value	Meaning
0	Automatic: let CPLEX choose; default
1	CPU time
2	Wall clock time (total physical time elapsed)

coefficient reduction setting

Purpose

Coefficient reduction setting

Syntax

C Name	CPX_PARAM_COEREDIND (int)
C++ Name	CoeRedInd (int)
Java Name	CoeRedInd (int)
.NET Name	CoeRedInd (int)
OPL Name	coeredind
Interactive Optimizer	preprocessing coeffreduce
Identifier	2004

Description

Decides how coefficient reduction is used. Coefficient reduction improves the objective value of the initial (and subsequent) LP relaxations solved during branch & cut by reducing the number of non-integral vertices.

Value Meaning

0	Do not use coefficient reduction
1	Reduce only to integral coefficients
2	Reduce all potential coefficients; default

variable (column) read limit

Purpose

Variable (column) read limit

Syntax

C Name	CPX_PARAM_COLREADLIM (int)
C++ Name	ColReadLim (int)
Java Name	ColReadLim (int)
.NET Name	ColReadLim (int)
Interactive Optimizer	read variables
Identifier	1023

Description

Specifies a limit for the number of columns (variables) to read for an allocation of memory.

This parameter does not restrict the size of a problem. Rather, it indirectly specifies the default amount of memory that will be pre-allocated before a problem is read from a file. If the limit is exceeded, more memory is automatically allocated.

Values

Any integer from 0 to 268 435 450; **default:** 60 000.

conflict information display

Purpose

Conflict information display

Syntax

C Name	CPX_PARAM_CONFLICTDISPLAY (int)
C++ Name	ConflictDisplay (int)
Java Name	ConflictDisplay (int)
.NET Name	ConflictDisplay (int)
OPL Name	conflictdisplay
Interactive Optimizer	conflict display i
Identifier	1074

Description

Decides how much information CPLEX reports when the conflict refiner is working.

Values

Value	Meaning
0	No display
1	Summary display; default
2	Detailed display

MIP covers switch

Purpose

MIP covers switch

Syntax

C Name	CPX_PARAM_COVERS (int)
C++ Name	Covers (int)
Java Name	Covers (int)
.NET Name	Covers (int)
OPL Name	covers
Interactive Optimizer	mip cuts covers
Identifier	2005

Description

Decides whether or not cover cuts should be generated for the problem. Setting the value to 0 (zero), the default, indicates that the attempt to generate covers should continue only if it seems to be helping.

Values

Value	Meaning
-1	Do not generate cover cuts
0	Automatic: let CPLEX choose; default
1	Generate cover cuts moderately
2	Generate cover cuts aggressively
3	Generate cover cuts very aggressively

simplex crash ordering

Purpose

Simplex crash ordering

Syntax

C Name	CPX_PARAM_CRAIND (int)
C++ Name	CraInd (int)
Java Name	CraInd (int)
.NET Name	CraInd (int)
OPL Name	craind
Interactive Optimizer	simplex crash
Identifier	1007

Description

Decides how CPLEX orders variables relative to the objective function when selecting an initial basis.

Values

Value	Meaning
-------	---------

LP Primal

-1	Alternate ways of using objective coefficients
0	Ignore objective coefficients during crash
1	Alternate ways of using objective coefficients; default

LP Dual

-1	Aggressive starting basis
0	Aggressive starting basis
1	Default starting basis; default

QP Primal

-1	Slack basis
0	Ignore Q terms and use LP solver for crash
1	Ignore objective and use LP solver for crash; default

Value Meaning

QP Dual

-1	Slack basis
0	Use Q terms for crash
1	Use Q terms for crash; default

lower cutoff

Purpose

Lower cutoff

Syntax

C Name	CPX_PARAM_CUTLO (double)
C++ Name	CutLo (double)
Java Name	CutLo (double)
.NET Name	CutLo (double)
OPL Name	cutlo
Interactive Optimizer	mip tolerances lowercutoff
Identifier	2006

Description

Sets the lower cutoff tolerance. When the problem is a maximization problem, CPLEX cuts off or discards solutions that are less than the specified cutoff value. If the model has no solution with an objective value greater than or equal to the cutoff value, then CPLEX declares the model infeasible. In other words, setting the lower cutoff value c for a maximization problem is similar to adding this constraint to the objective function of the model: $obj \geq c$.

Tip: This parameter is not effective with the conflict refiner nor with FeasOpt. That is, neither of those tools can analyze an infeasibility introduced by this parameter. If you want to analyze such a condition, add an explicit objective constraint to your model instead before you invoke either of those tools.

Values

Any number; **default:** -1e+75.

number of cutting plane passes

Purpose

Number of cutting plane passes

Syntax

C Name	CPX_PARAM_CUTPASS (int)
C++ Name	CutPass (int)
Java Name	CutPass (int)
.NET Name	CutPass (int)
OPL Name	cutpass
Interactive Optimizer	mip limits cutpasses
Identifier	2056

Description

Sets the upper limit on the number of cutting plane passes CPLEX performs when solving the root node of a MIP model.

Values

Value	Meaning
-1	None
0	Automatic: let CPLEX choose; default
Any positive integer	Number of passes to perform

row multiplier factor for cuts

Purpose

Row multiplier factor for cuts

Syntax

C Name	CPX_PARAM_CUTSFACTOR (double)
C++ Name	CutsFactor (double)
Java Name	CutsFactor (double)
.NET Name	CutsFactor (double)
OPL Name	cutsfactor
Interactive Optimizer	mip limits cutsfactor
Identifier	2033

Description

Limits the number of cuts that can be added. The number of rows in the problem with cuts added is limited to `CutsFactor` times the original number of rows. If the problem is presolved, the original number of rows is that from the presolved problem.

A `CutsFactor` of 1.0 or less means that no cuts will be generated.

Because cuts can be added and removed during the course of optimization, `CutsFactor` may not correspond directly to the number of cuts seen in the node log or in the summary table at the end of optimization.

Values

Any nonnegative number; **default:** 4.0

upper cutoff

Purpose

Upper cutoff

Syntax

C Name	CPX_PARAM_CUTUP (double)
C++ Name	CutUp (double)
Java Name	CutUp (double)
.NET Name	CutUp (double)
OPL Name	cutup
Interactive Optimizer	mip tolerances uppercutoff
Identifier	2007

Description

Sets the upper cutoff tolerance. When the problem is a minimization problem, CPLEX cuts off or discards any solutions that are greater than the specified upper cutoff value. If the model has no solution with an objective value less than or equal to the cutoff value, CPLEX declares the model infeasible. In other words, setting an upper cutoff value c for a minimization problem is similar to adding this constraint to the objective function of the model: $obj \leq c$.

Tip: This parameter is not effective with the conflict refiner nor with FeasOpt. That is, neither of those tools can analyze an infeasibility introduced by this parameter. If you want to analyze such a condition, add an explicit objective constraint to your model instead before you invoke either of those tools.

Values

Any number; **default:** 1e+75.

data consistency checking switch

Purpose

Data consistency checking switch

Syntax

C Name	CPX_PARAM_DATACHECK (int)
C++ Name	DataCheck (bool)
Java Name	DataCheck (bool)
.NET Name	DataCheck (bool)
OPL Name	datacheck
Interactive Optimizer	read datacheck
Identifier	1056

Description

Decides whether data should be checked for consistency. When this parameter is on, the routines `CPXcopy_____`, `CPXread_____` and `CPXchg_____` of the C API perform extensive checking of data in their array arguments, such as checking that indices are within range, that there are no duplicate entries, and that values are valid for the type of data or are valid numbers. This checking is useful for debugging applications. When this checking identifies trouble, you can gather more specific detail by calling one of the routines in `check.c`.

Values

int	bool	Symbol	Meaning
0	false	CPX_OFF	Data checking off; do not check; default
1	true	CPX_ON	Data checking on

dependency switch

Purpose

Dependency switch

Syntax

C Name	CPX_PARAM_DEPIND (int)
C++ Name	DepInd (int)
Java Name	DepInd (int)
.NET Name	DepInd (int)
OPL Name	depind
Interactive Optimizer	preprocessing dependency
Identifier	1008

Description

Decides whether to activate the dependency checker. If on, the dependency checker searches for dependent rows during preprocessing. If off, dependent rows are not identified.

Values

Value	Meaning
-1	Automatic: let CPLEX choose; default
0	Off: do not use dependency checker
1	Turn on only at the beginning of preprocessing
2	Turn on only at the end of preprocessing
3	Turn on at the beginning and at the end of preprocessing

MIP disjunctive cuts switch

Purpose

MIP disjunctive cuts switch

Syntax

C Name	CPX_PARAM_DISJCUTS (int)
C++ Name	DisjCuts (int)
Java Name	DisjCuts (int)
.NET Name	DisjCuts (int)
OPL Name	disjcuts
Interactive Optimizer	mip cuts disjunctive
Identifier	2053

Description

Decides whether or not disjunctive cuts should be generated for the problem. Setting the value to 0 (zero), the default, indicates that the attempt to generate disjunctive cuts should continue only if it seems to be helping.

Values

Value	Meaning
-1	Do not generate disjunctive cuts
0	Automatic: let CPLEX choose; default
1	Generate disjunctive cuts moderately
2	Generate disjunctive cuts aggressively
3	Generate disjunctive cuts very aggressively

MIP dive strategy

Purpose

MIP dive strategy

Syntax

C Name	CPX_PARAM_DIVETYPE (int)
C++ Name	DiveType (int)
Java Name	DiveType (int)
.NET Name	DiveType (int)
OPL Name	divetype
Interactive Optimizer	mip strategy dive
Identifier	2060

Description

Controls the MIP dive strategy. The MIP traversal strategy occasionally performs probing dives, where it looks ahead at both children nodes before deciding which node to choose. The default (automatic) setting lets CPLEX choose when to perform a probing dive, 1 (one) directs CPLEX never to perform probing dives, 2 always to probe, 3 to spend more time exploring potential solutions that are similar to the current incumbent. Setting 2, always to probe, is helpful for finding integer solutions.

Values

Value	Meaning
0	Automatic: let CPLEX choose; default
1	Traditional dive
2	Probing dive
3	Guided dive

dual simplex pricing algorithm

Purpose

Dual simplex pricing algorithm

Syntax

C Name	CPX_PARAM_DPRIIND (int)
C++ Name	DPriInd (int)
Java Name	DPriInd (int)
.NET Name	DPriInd (int)
OPL Name	dpriind
Interactive Optimizer	simplex dgradient
Identifier	1009

Description

Decides the type of pricing applied in the dual simplex algorithm. The default pricing (0) usually provides the fastest solution time, but many problems benefit from alternate settings.

Values

Value	Symbol	Meaning
0	CPX_DPRIIND_AUTO	Automatic: let CPLEX choose; default
1	CPX_DPRIIND_FULL	Standard dual pricing
2	CPX_DPRIIND_STEEP	Steepest-edge pricing
3	CPX_DPRIIND_FULL_STEEP	Steepest-edge pricing in slack space
4	CPX_DPRIIND_STEEPQSTART	Steepest-edge pricing, unit initial norms
5	CPX_DPRIIND_DEVEX	devex pricing

See also

candidate limit for generating Gomory fractional cuts, MIP Gomory fractional cuts switch, pass limit for generating Gomory fractional cuts

type of cut limit

Purpose

Type of cut limit

Syntax

C Name	CPX_PARAM_EACHCUTLIM (int)
C++ Name	EachCutLim (int)
Java Name	EachCutLim (int)
.NET Name	EachCutLim (int)
OPL Name	eachcutlim
Interactive Optimizer	mip limit eachcutlimit
Identifier	2102

Description

Sets a limit for each type of cut.

This parameter allows you to set a uniform limit on the number of cuts of each type that CPLEX generates. By default, the limit is the largest integer supported by a given platform; that is, there is no effective limit by default.

Tighter limits on the number of cuts of each type may benefit certain models. For example, a limit on each type of cut will prevent any one type of cut from being created in such large number that the limit on the total number of all types of cuts is reached before other types of cuts have an opportunity to be created.

A setting of 0 (zero) means no cuts.

This parameter does **not** influence the number of Gomory cuts. For means to control the number of Gomory cuts, see also the fractional cut parameters:

- ◆ *candidate limit for generating Gomory fractional cuts*: CPX_PARAM_FRACCAND, FracCand;
- ◆ *MIP Gomory fractional cuts switch*: CPX_PARAM_FRACCUTS, FracCuts;
- ◆ *pass limit for generating Gomory fractional cuts*: CPX_PARAM_FRACPASS, FracPass.

Values

Value	Meaning
0	No cuts
Any positive number	Limit each type of cut
2100000000	default

absolute MIP gap tolerance

Purpose

Absolute MIP gap tolerance

Syntax

C Name	CPX_PARAM_EPAGAP (double)
C++ Name	EpAGap (double)
Java Name	EpAGap (double)
.NET Name	EpAGap (double)
OPL Name	epagap
InteractiveOptimizer	mip tolerances absmipgap
Identifier	2008

Description

Sets an absolute tolerance on the gap between the best integer objective and the objective of the best node remaining. When this difference falls below the value of this parameter, the mixed integer optimization is stopped.

Values

Any nonnegative number; **default:** 1e-06.

relative MIP gap tolerance

Purpose

Relative MIP gap tolerance

Syntax

C Name	CPX_PARAM_EPGAP (double)
C++ Name	EpGap (double)
Java Name	EpGap (double)
.NET Name	EpGap (double)
OPL Name	epgap
Interactive Optimizer	mip tolerances mipgap
Identifier	2009

Description

When the value

$$\frac{|\text{bestnode} - \text{bestinteger}|}{(1e-10 + |\text{bestinteger}|)}$$

falls below the value of this parameter, the mixed integer optimization is stopped.

For example, to instruct CPLEX to stop as soon as it has found a feasible integer solution proved to be within five percent of optimal, set the relative mipgap tolerance to 0.05.

Values

Any number from 0.0 to 1.0; **default:** 1e-04.

integrality tolerance

Purpose

Integrality tolerance

Syntax

C Name	CPX_PARAM_EPINT (double)
C++ Name	EpInt (double)
Java Name	EpInt (double)
.NET Name	EpInt (double)
OPL Name	epint
Interactive Optimizer	mip tolerances integrality
Identifier	2010

Description

Specifies the amount by which an integer variable can be different from an integer and still be considered feasible.

A value of zero is permitted, and the optimizer will attempt to meet this tolerance.

However, in some models, computer roundoff may still result in small, nonzero deviations from integrality. If any of these deviations exceed the value of this parameter, or exceed $1e-10$ in the case where this parameter has been set to a value less than that, a solution status of CPX_STAT_OPTIMAL_INFEAS will be returned instead of the usual CPX_STAT_OPTIMAL.

Values

Any number from 0.0 to 0.5; **default:** 1e-05.

epsilon used in linearization

Purpose

Epsilon used in linearization

Syntax

C Name	CPX_PARAM_EPLIN but not applicable in the C API
C++ Name	EpLin (double)
Java Name	EpLin (double)
.NET Name	EpLin (double)
Interactive Optimizer	not available in the Interactive Optimizer
Identifier	2068

Description

Sets the epsilon (degree of tolerance) used in linearization in the object-oriented APIs.

Not applicable in the C API.

Not available in the Interactive Optimizer.

This parameter controls how strict inequalities are managed during linearization. In other words, it provides an epsilon for deciding when two values are not equal during linearization. For example, when x is a numeric variable (that is, an instance of `IloNumVar`),

$x < a$

becomes

$x \leq a - \text{eplin}$.

Similarly, $x \neq a$

becomes

$\{(x < a) \ || \ (x > a)\}$

which is linearized automatically for you in the object-oriented APIs as

$\{(x \leq a - \text{eplin}) \ || \ (x \geq a + \text{eplin})\}$.

Exercise caution in changing this parameter from its default value: the smaller the epsilon, the more numerically unstable the model will tend to become. If you are not getting an expected

solution for an object-oriented model that uses linearization, it might be that this solution is cut off because of the relatively high `EpLin` value. In such a case, carefully try reducing it.

Values

Any positive value greater than zero; **default:** 1e-3.

Markowitz tolerance

Purpose

Markowitz tolerance

Syntax

C Name	CPX_PARAM_EPMRK (double)
C++ Name	EpMrk (double)
Java Name	EpMrk (double)
.NET Name	EpMrk (double)
OPL Name	epmrk
Interactive Optimizer	simplex tolerances markowitz
Identifier	1013

Description

Influences pivot selection during basis factoring. Increasing the Markowitz threshold may improve the numerical properties of the solution.

Values

Any number from 0.0001 to 0.99999; **default**: 0.01.

optimality tolerance

Purpose

Optimality tolerance

Syntax

C Name	CPX_PARAM_EPOPT (double)
C++ Name	EpOpt (double)
Java Name	EpOpt (double)
.NET Name	EpOpt (double)
OPL Name	epopt
Interactive Optimizer	simplex tolerances optimality
Identifier	1014

Description

Influences the reduced-cost tolerance for optimality. This parameter governs how closely CPLEX must approach the theoretically optimal solution.

Values

Any number from 1e-9 to 1e-1; **default:** 1e-06.

perturbation constant

Purpose

Perturbation constant

Syntax

C Name	CPX_PARAM_EPPER (double)
C++ Name	EpPer (double)
Java Name	EpPer (double)
.NET Name	EpPer (double)
OPL Name	epper
Interactive Optimizer	simplex perturbation
Identifier	1015

Description

Sets the amount by which CPLEX perturbs the upper and lower bounds or objective coefficients on the variables when a problem is perturbed in the simplex algorithm. This parameter can be set to a smaller value if the default value creates too large a change in the problem.

Values

Any positive number greater than or equal to 1e-8; **default:** 1e-6.

relaxation for FeasOpt

Purpose

Relaxation for `feasOpt`

Syntax

C Name	<code>CPX_PARAM_EPRELAX</code> (double)
C++ Name	<code>EpRelax</code> (double)
Java Name	<code>EpRelax</code> (double)
.NET Name	<code>EpRelax</code> (double)
OPL Name	<code>eprelax</code>
Interactive Optimizer	<code>feasopt tolerance</code>
Identifier	2073

Description

Controls the amount of relaxation for the routine `CPXfeasopt` in the C API or for the method `feasOpt` in the object-oriented APIs.

In the case of a MIP, it serves the purpose of the absolute gap for the `feasOpt` model in Phase I (the phase to minimize relaxation).

Using this parameter, you can implement other stopping criteria as well. To do so, first call `feasOpt` with the stopping criteria that you prefer; then set this parameter to the resulting objective of the Phase I model; unset the other stopping criteria, and call `feasOpt` again. Since the solution from the first call already matches this parameter, Phase I will terminate immediately in this second call to `feasOpt`, and Phase II will start.

In the case of an LP, this parameter controls the lower objective limit for Phase I of `feasOpt` and is thus relevant only when the primal optimizer is in use.

Values

Any nonnegative value; **default:** 1e-6.

See also

lower objective value limit

feasibility tolerance

Purpose

Feasibility tolerance

Syntax

C Name	CPX_PARAM_EPRHS (double)
C++ Name	EpRHS (double)
Java Name	EpRHS (double)
.NET Name	EpRHS (double)
OPL Name	eprhs
Interactive Optimizer	simplex tolerances feasibility
Identifier	1016

Description

Specifies the feasibility tolerance, that is, the degree to which values of the basic variables calculated by the simplex method may violate their bounds. Feasibility influences the selection of an optimal basis and can be reset to a higher value when a problem is having difficulty maintaining feasibility during optimization. You may also wish to lower this tolerance after finding an optimal solution if there is any doubt that the solution is truly optimal. If the feasibility tolerance is set too low, CPLEX may falsely conclude that a problem is infeasible. If you encounter reports of infeasibility during Phase II of the optimization, a small adjustment in the feasibility tolerance may improve performance.

Values

Any number from 1e-9 to 1e-1; **default:** 1e-06.

mode of FeasOpt

Purpose

Mode of FeasOpt

Syntax

C Name	CPX_PARAM_FEASOPTMODE (int)
C++ Name	FeasOptMode (int)
Java Name	FeasOptMode (int)
.NET Name	FeasOptMode (int)
OPL Name	feasoptmode
Interactive Optimizer	feasopt mode
Identifier	1084

Description

Decides how FeasOpt measures the relaxation when finding a minimal relaxation in an infeasible model. FeasOpt works in two phases. In its first phase, it attempts to minimize its relaxation of the infeasible model. That is, it attempts to find a feasible solution that requires minimal change. In its second phase, it finds an optimal solution among those that require only as much relaxation as it found necessary in the first phase. Values of this parameter indicate two aspects to CPLEX:

- ◆ whether to stop in phase one or continue to phase two and
- ◆ how to measure the relaxation, according to one of the following criteria:
 - as a sum of required relaxations;
 - as the number of constraints and bounds required to be relaxed;
 - as a sum of the squares of required relaxations.

Values

Value	Symbol	Symbol (C API)	Meaning
0	MinSum	CPX_FEASOPT_MIN_SUM	Minimize the sum of all required relaxations in first phase only; default

Value	Symbol	Symbol (C API)	Meaning
1	OptSum	CPX_FEASOPT_OPT_SUM	Minimize the sum of all required relaxations in first phase and execute second phase to find optimum among minimal relaxations
2	MinInf	CPX_FEASOPT_MIN_INF	Minimize the number of constraints and bounds requiring relaxation in first phase only
3	OptInf	CPX_FEASOPT_OPT_INF	Minimize the number of constraints and bounds requiring relaxation in first phase and execute second phase to find optimum among minimal relaxations
4	MinQuad	CPX_FEASOPT_MIN_QUAD	Minimize the sum of squares of required relaxations in first phase only
5	OptQuad	CPX_FEASOPT_OPT_QUAD	Minimize the sum of squares of required relaxations in first phase and execute second phase to find optimum among minimal relaxations

MIP flow cover cuts switch

Purpose

MIP flow cover cuts switch

Syntax

C Name	CPX_PARAM_FLOWCOVERS (int)
C++ Name	FlowCovers (int)
Java Name	FlowCovers (int)
.NET Name	FlowCovers (int)
OPL Name	flowcovers
Interactive Optimizer	mip cuts flowcovers
Identifier	2040

Description

Decides whether or not to generate flow cover cuts for the problem. Setting the value to 0 (zero), the default, indicates that the attempt to generate flow cover cuts should continue only if it seems to be helping.

Values

Value	Meaning
-1	Do not generate flow cover cuts
0	Automatic: let CPLEX choose; default
1	Generate flow cover cuts moderately
2	Generate flow cover cuts aggressively

MIP flow path cut switch

Purpose

MIP flow path cut switch

Syntax

C Name	CPX_PARAM_FLOWPATHS (int)
C++ Name	FlowPaths (int)
Java Name	FlowPaths (int)
.NET Name	FlowPaths (int)
OPL Name	flowpaths
Interactive Optimizer	mip cuts pathcut
Identifier	2051

Description

Decides whether or not flow path cuts should be generated for the problem. Setting the value to 0 (zero), the default, indicates that the attempt to generate flow path cuts should continue only if it seems to be helping.

Values

Value	Meaning
-1	Do not generate flow path cuts
0	Automatic: let CPLEX choose; default
1	Generate flow path cuts moderately
2	Generate flow path cuts aggressively

feasibility pump switch

Purpose

Feasibility pump switch

Syntax

C Name	CPX_PARAM_FPHEUR (int)
C++ Name	FPHeur (int)
Java Name	FPHeur (int)
.NET Name	FPHeur (int)
OPL Name	fpheur
Interactive Optimizer	mip strategy fpheur
Identifier	2098

Description

Turns on or off the feasibility pump heuristic for mixed integer programming (MIP) models.

At the default setting 0 (zero), CPLEX automatically chooses whether or not to apply the feasibility pump heuristic on the basis of characteristics of the model. The feasibility pump does **not** apply to models of the type mixed integer quadratically constrained programs (MIQCP).

To turn off the feasibility pump heuristic, set the parameter to -1 (minus one).

To turn on the feasibility pump heuristic, set the parameter to 1 (one) or 2.

If the parameter is set to 1 (one), the feasibility pump tries to find a feasible solution without taking the objective function into account.

If the parameter is set to 2, the heuristic usually finds solutions of better objective value, but is more likely to fail to find a feasible solution.

For more detail about the feasibility pump heuristic, see research by Fischetti, Glover, and Lodi (2003, 2005), by Bertacco, Fischetti, and Lodi (2005), and by Achterberg and Berthold (2005, 2007).

Values

Value Meaning

-1	Do not apply the feasibility pump heuristic
0	Automatic: let CPLEX choose; default
1	Apply the feasibility pump heuristic with an emphasis on finding a feasible solution
2	Apply the feasibility pump heuristic with an emphasis on finding a feasible solution with a good objective value

candidate limit for generating Gomory fractional cuts

Purpose

Candidate limit for generating Gomory fractional cuts

Syntax

C Name	CPX_PARAM_FRACCAND (int)
C++ Name	FracCand (int)
Java Name	FracCand (int)
.NET Name	FracCand (int)
OPL Name	fraccand
Interactive Optimizer	mip limits gomorycand
Identifier	2048

Description

Limits the number of candidate variables for generating Gomory fractional cuts.

Values

Any positive integer; **default**: 200.

MIP Gomory fractional cuts switch

Purpose

MIP Gomory fractional cuts switch

Syntax

C Name	CPX_PARAM_FRACCUTS (int)
C++ Name	FracCuts (int)
Java Name	FracCuts (int)
.NET Name	FracCuts (int)
OPL Name	fraccuts
Interactive Optimizer	mip cuts gomory
Identifier	2049

Description

Decides whether or not Gomory fractional cuts should be generated for the problem. Setting the value to 0 (zero), the default, indicates that the attempt to generate Gomory fractional cuts should continue only if it seems to be helping.

Values

Value	Meaning
-1	Do not generate Gomory fractional cuts
0	Automatic: let CPLEX choose; default
1	Generate Gomory fractional cuts moderately
2	Generate Gomory fractional cuts aggressively

pass limit for generating Gomory fractional cuts

Purpose

Pass limit for generating Gomory fractional cuts

Syntax

C Name	CPX_PARAM_FRACPASS (int)
C++ Name	FracPass (int)
Java Name	FracPass (int)
.NET Name	FracPass (int)
OPL Name	fracpass
Interactive Optimizer	mip limits gomorypass
Identifier	2050

Description

Limits the number of passes for generating Gomory fractional cuts. At the default setting of 0 (zero), CPLEX decides the number of passes to make. The parameter is ignored if the Gomory fractional cut parameter (*MIP Gomory fractional cuts switch*: CPX_PARAM_FRACCUTS, FracCuts) is set to a nonzero value.

Values

Value	Meaning
0	Automatic: let CPLEX choose; default
Any positive integer	Number of passes to generate Gomory fractional cuts

MIP GUB cuts switch

Purpose

MIP GUB cuts switch

Syntax

C Name	CPX_PARAM_GUBCOVERS (int)
C++ Name	GUBCovers (int)
Java Name	GUBCovers (int)
.NET Name	GUBCovers (int)
OPL Name	gubcovers
Interactive Optimizer	mip cuts gubcovers
Identifier	2044

Description

Decides whether or not to generate GUB cuts for the problem. Setting the value to 0 (zero), the default, indicates that the attempt to generate GUB cuts should continue only if it seems to be helping.

Values

Value	Meaning
-1	Do not generate GUB cuts
0	Automatic: let CPLEX choose; default
1	Generate GUB cuts moderately
2	Generate GUB cuts aggressively

MIP heuristic frequency

Purpose

MIP heuristic frequency

Syntax

C Name	CPX_PARAM_HEURFREQ (int)
C++ Name	HeurFreq (int)
Java Name	HeurFreq (int)
.NET Name	HeurFreq (int)
OPL Name	heurfreq
Interactive Optimizer	mip strategy heuristicfreq
Identifier	2031

Description

Decides how often to apply the periodic heuristic. Setting the value to -1 turns off the periodic heuristic. Setting the value to 0 (zero), the default, applies the periodic heuristic at an interval chosen automatically. Setting the value to a positive number applies the heuristic at the requested node interval. For example, setting this parameter to 20 dictates that the heuristic be called at node 0, 20, 40, 60, etc.

Values

Value	Meaning
-1	None
0	Automatic: let CPLEX choose; default
Any positive integer	Apply the periodic heuristic at this frequency

MIP implied bound cuts switch

Purpose

MIP implied bound cuts switch

Syntax

C Name	CPX_PARAM_IMPLBD (int)
C++ Name	ImplBd (int)
Java Name	ImplBd (int)
.NET Name	ImplBd (int)
OPL Name	implbd
Interactive Optimizer	mip cuts implied
Identifier	2041

Description

Decides whether or not to generate implied bound cuts for the problem. Setting the value to 0 (zero), the default, indicates that the attempt to generate implied bound cuts should continue only if it seems to be helping.

Values

Value	Meaning
-1	Do not generate implied bound cuts
0	Automatic: let CPLEX choose; default
1	Generate implied bound cuts moderately
	Generate implied bound cuts aggressively

MIP integer solution limit

Purpose

MIP integer solution limit

Syntax

C Name	CPX_PARAM_INTSOLLIM (int)
C++ Name	IntSolLim (int)
Java Name	IntSolLim (int)
.NET Name	IntSolLim (int)
OPL Name	intsollim
Interactive Optimizer	mip limits solutions
Identifier	2015

Description

Sets the number of MIP solutions to be found before stopping.

This integer solution limit does **not** apply to the populate procedure, which generates solutions to store in the solution pool. For a limit on the number of solutions generated by populate, see the populate limit parameter: *limit on number of solutions generated for solution pool*.

Values

Any positive integer strictly greater than zero; zero is **not** allowed; **default**: 2100000000.

See also

limit on number of solutions generated for solution pool

simplex maximum iteration limit

Purpose

Simplex maximum iteration limit

Syntax

C Name	CPX_PARAM_ITLIM (int)
C++ Name	ItLim (int)
Java Name	ItLim (int)
.NET Name	ItLim (int)
OPL Name	itlim
Interactive Optimizer	simplex limits iterations
Identifier	1020

Description

Sets the maximum number of simplex iterations to be performed before the algorithm terminates without reaching optimality. When set to 0 (zero), no simplex method iteration occurs. However, CPLEX factors the initial basis from which solution routines provide information about the associated initial solution.

Values

Any nonnegative integer; **default:** 2100000000.

local branching heuristic

Purpose

Local branching heuristic

Syntax

C Name	CPX_PARAM_LBHEUR (int)
C++ Name	LBHeur (bool)
Java Name	LBHeur (bool)
.NET Name	LBHeur (bool)
OPL Name	lbheur
Interactive Optimizer	mip strategy lbheur
Identifier	2063

Description

Controls whether CPLEX applies a local branching heuristic to try to improve new incumbents found during a MIP search. By default, this parameter is off. If you turn it on, CPLEX will invoke a local branching heuristic only when it finds a new incumbent. If CPLEX finds multiple incumbents at a single node, the local branching heuristic will be applied only to the last one found.

Values

Value	bool	Symbol	Meaning
0	false	CPX_OFF	Local branching heuristic is off; default
1	true	CPX_ON	Apply local branching heuristic to new incumbent

memory reduction switch

Purpose

Reduces use of memory

Syntax

C Name	CPX_PARAM_MEMORYEMPHASIS (int)
C++ Name	MemoryEmphasis (bool)
Java Name	MemoryEmphasis (bool)
.NET Name	MemoryEmphasis (bool)
OPL Name	memoryemphasis
Interactive Optimizer	emphasis memory
Identifier	1082

Description

Directs CPLEX that it should conserve memory where possible. When you set this parameter to its nondefault value, CPLEX will choose tactics, such as data compression or disk storage, for some of the data computed by the simplex, barrier, and MIP optimizers. Of course, conserving memory may impact performance in some models. Also, while solution information will be available after optimization, certain computations that require a basis that has been factored (for example, for the computation of the condition number Kappa) may be unavailable.

Values

Value	bool	Symbol	Meaning
0	false	CPX_OFF	Off; do not conserve memory; default
1	true	CPX_ON	On; conserve memory where possible

MIP callback switch between original model and reduced, presolved model

Purpose

MIP callback switch between original model and reduced, presolved model

Syntax

C Name	CPX_PARAM_MIPCBREDLP (int)
C++ Name	MIP callback reduced LP parameter not available in this API
Java Name	not available
.NET Name	not available
Interactive Optimizer	not available
Identifier	2055

Description

Controls whether your callback accesses node information of the original model (off) or node information of the reduced, presolved model (on, default). Advanced routines to control MIP callbacks (such as `CPXgetcallbacklp`, `CPXsetheuristiccallbackfunc`, `CPXsetbranchcallbackfunc`, `CPXgetbranchcallbackfunc`, `CPXsetcutcallbackfunc`, `CPXsetincumbentcallbackfunc`, `CPXgetcallbacksosinfo`, `CPXcutcallbackadd`, `CPXcutcallbackaddlocal`, and others) consider the setting of this parameter and access the original model or the reduced, presolved model accordingly.

The routine `CPXgetcallbacknodeip` is an exception: it always accesses the current node LP associated with the presolved model, regardless of the setting of this parameter.

For certain routines, such as `CPXcutcallbackadd`, when you set the parameter `CPX_PARAM_MIPCBREDLP` to zero, you should also set `CPX_PARAM_PRELINEAR` to zero as well.

In the C++, Java, and .NET APIs of CPLEX, only the original model is available to callbacks. In other words, this parameter is effective only for certain advanced routines of the C API.

Values

Value	Symbol	Meaning
0	CPX_OFF	Off: use original model
1	CPX_ON	On: use reduced, presolved model; default

MIP node log display information

Purpose

MIP node log display information

Syntax

C Name	CPX_PARAM_MIPDISPLAY (int)
C++ Name	MIPDisplay (int)
Java Name	MIPDisplay (int)
.NET Name	MIPDisplay (int)
OPL Name	mipdisplay
Interactive Optimizer	mip display
Identifier	2012

Description

Decides what CPLEX reports to the screen during mixed integer optimization (MIP).

The amount of information displayed increases with increasing values of this parameter.

◆ A setting of 0 (zero) causes no node log to be displayed until the **optimal solution** is found.

◆ A setting of 1 (one) displays an entry for each **integer feasible solution** found.

Each entry contains:

- the value of the **objective function**;
- the **node count**;
- the **number of unexplored nodes** in the tree;
- the current **optimality gap**.

◆ A setting of 2 also generates an entry for every **n-th node** (where n is the setting of the *MIP node log interval* parameter).

◆ A setting of 3 additionally generates an entry for every n-th node giving the number of **cuts** added to the problem for the previous `MIPInterval` number of nodes, plus an entry for each successfully processed **MIP start**.

- ◆ A setting of 4 additionally generates entries for the **LP root relaxation** according to the setting of the parameter to control the *simplex iteration information display* (SimDisplay, CPX_PARAM_SIMDISPLAY).
- ◆ A setting of 5 additionally generates entries for the **LP subproblems**, also according to the setting of the parameter to control the *simplex iteration information display* (SimDisplay, CPX_PARAM_SIMDISPLAY).

Values

Value	Meaning
0	No display until optimal solution has been found
1	Display integer feasible solutions
2	Display integer feasible solutions plus an entry for every n-th node; default
3	Display integer feasible solutions, every n-th node entry, number of cuts added, and information about the processing of each successful MIP start
4	Display integer feasible solutions, every n-th node entry, number of cuts added, information about the processing of each successful MIP start, and information about the LP subproblem at root
5	Display integer feasible solutions, every n-th node entry, number of cuts added, information about the processing of each successful MIP start, and information about the LP subproblem at root and at nodes

See also

MIP node log interval, simplex iteration information display, network logging display switch, and messages to screen switch

MIP emphasis switch

Purpose

MIP emphasis switch

Syntax

C Name	CPX_PARAM_MIPEMPHASIS (int)
C++ Name	MIPEmphasis (int)
Java Name	MIPEmphasis (int)
.NET Name	MIPEmphasis (int)
OPL Name	mipemphasis
Interactive Optimizer	emphasis mip
Identifier	2058

Description

Controls trade-offs between speed, feasibility, optimality, and moving bounds in MIP.

With the default setting of **BALANCED**, CPLEX works toward a rapid proof of an optimal solution, but balances that with effort toward finding high quality feasible solutions early in the optimization.

When this parameter is set to **FEASIBILITY**, CPLEX frequently will generate more feasible solutions as it optimizes the problem, at some sacrifice in the speed to the proof of optimality.

When set to **OPTIMALITY**, less effort may be applied to finding feasible solutions early.

With the setting **BESTBOUND**, even greater emphasis is placed on proving optimality through moving the best bound value, so that the detection of feasible solutions along the way becomes almost incidental.

When the parameter is set to **HIDDENFEAS**, the MIP optimizer works hard to find high quality feasible solutions that are otherwise very difficult to find, so consider this setting when the **FEASIBILITY** setting has difficulty finding solutions of acceptable quality.

Values

Value	Symbol	Meaning
0	CPX_MIPEMPHASIS_BALANCED	Balance optimality and feasibility; default
1	CPX_MIPEMPHASIS_FEASIBILITY	Emphasize feasibility over optimality

Value	Symbol	Meaning
2	CPX_MIPEMPHASIS_OPTIMALITY	Emphasize optimality over feasibility
3	CPX_MIPEMPHASIS_BESTBOUND	Emphasize moving best bound
4	CPX_MIPEMPHASIS_HIDDENFEAS	Emphasize finding hidden feasible solutions

MIP node log interval

Purpose

MIP node log interval

Syntax

C Name	CPX_PARAM_MIPINTERVAL (int)
C++ Name	MIPInterval (int)
Java Name	MIPInterval (int)
.NET Name	MIPInterval (int)
OPL Name	mipinterval
Interactive Optimizer	mip interval
Identifier	2013

Description

Controls the frequency of node logging when the MIP display parameter (*MIP node log display information*) is set higher than 1 (one).

Values

Any positive integer; **default**: 100.

See also

MIP node log display information

MIP priority order switch

Purpose

MIP priority order switch

Syntax

C Name	CPX_PARAM_MIPORDIND (int)
C++ Name	MIPOrdInd (bool)
Java Name	MIPOrdInd (bool)
.NET Name	MIPOrdInd (bool)
OPL Name	mipordind
Interactive Optimizer	mip strategy order
Identifier	2020

Description

Decides whether to use the priority order, if one exists, for the next mixed integer optimization.

Values

Value	bool	Symbol	Meaning
false	CPX_OFF	Off: do not use priority order	
true	CPX_ON	On: use priority order, if it exists; default	

MIP priority order generation

Purpose

MIP priority order generation

Syntax

C Name	CPX_PARAM_MIPORDTYPE (int)
C++ Name	MIPOrdType (int)
Java Name	MIPOrdType (int)
.NET Name	MIPOrdType (int)
OPL Name	mipordtype
Interactive Optimizer	mip ordertype
Identifier	2032

Description

Selects the type of generic priority order to generate when no priority order is present.

Values

Value	Symbol	Meaning
0	default	Do not generate a priority order
1	CPX_MIPORDER_COST	Use decreasing cost
2	CPX_MIPORDER_BOUNDS	Use increasing bound range
3	CPX_MIPORDER_SCALED COST	Use increasing cost per coefficient count

MIP dynamic search switch

Purpose

MIP dynamic search switch

Syntax

C Name	CPX_PARAM_MIPSEARCH (int)
C++ Name	MIPSearch (int)
Java Name	MIPSearch (int)
.NET Name	MIPSearch (int)
OPL Name	mipsearch
Interactive Optimizer	mip strategy search
Identifier	2109

Description

Sets the search strategy for a mixed integer program (MIP). By default, CPLEX chooses whether to apply dynamic search or conventional branch and cut based on characteristics of the model and the presence (or absence) of callbacks.

Only informational callbacks are compatible with dynamic search. For more detail about informational callbacks and how to create and install them in your application, see *Informational callbacks* in the *ILOG CPLEX User's Manual*.

To benefit from dynamic search, a MIP must **not** include query callbacks. In other words, query callbacks are not compatible with dynamic search. For a more detailed definition of query or diagnostic callbacks, see *Query or diagnostic callbacks* in the *ILOG CPLEX User's Manual*.

To benefit from dynamic search, a MIP must **not** include control callbacks (that is, callbacks that alter the search path through the solution space). In other words, control callbacks are not compatible with dynamic search. These control callbacks are identified as **advanced** in the reference manuals of the APIs. If control callbacks are present in your application, CPLEX will disable dynamic search, issue a warning, and apply only static branch and cut. If you want to control the search yourself, for example, through advanced control callbacks, then you should set this parameter to 1 (one) to disable dynamic search and to apply conventional branch and cut.

Values

Value	Symbolic Name	Meaning
0	CPX_MIPSEARCH_AUTO	Automatic: let CPLEX choose; default
1	CPX_MIPSEARCH_TRADITIONAL	Apply traditional branch and cut strategy; disable dynamic search
2	CPX_MIPSEARCH_DYNAMIC	Apply dynamic search

MIQCP strategy switch

Purpose

MIQCP strategy switch

Syntax

C Name	CPX_PARAM_MIQCPSTRAT (int)
C++ Name	MIQCPStrat (int)
Java Name	MIQCPStrat (int)
.NET Name	MIQCPStrat (int)
OPL Name	miqcpstrat
Interactive Optimizer	mip strategy miqcpstrat
Identifier	2110

Description

Sets the strategy that CPLEX uses to solve a quadratically constrained mixed integer program (MIQCP).

This parameter controls how MIQCPs (that is, mixed integer programs with one or more constraints including quadratic terms) are solved. For more detail about the types of quadratically constrained models that CPLEX solves, see *Identifying a quadratically constrained program (QCP)* in the *CPLEX User's Manual*.

At the default setting of 0 (zero), CPLEX automatically chooses a strategy.

When you set this parameter to the value 1 (one), you tell CPLEX to solve a **QCP relaxation** of the model at each node.

When you set this parameter to the value 2, you tell CPLEX to attempt to solve an **LP relaxation** of the model at each node.

For some models, the setting 2 may be more effective than 1 (one). You may need to experiment with this parameter to determine the best setting for your model.

Values

Value	Meaning
0	Automatic: let CPLEX choose; default
1	Solve a QCP node relaxation at each node

Value Meaning

2	Solve an LP node relaxation at each node
---	--

MIP MIR (mixed integer rounding) cut switch

Purpose

MIP MIR (mixed integer rounding) cut switch

Syntax

C Name	CPX_PARAM_MIRCUTS (int)
C++ Name	MIRCuts (int)
Java Name	MIRCuts (int)
.NET Name	MIRCuts (int)
OPL Name	mircuts
Interactive Optimizer	mip cuts mircut
Identifier	2052

Description

Decides whether or not to generate MIR cuts (mixed integer rounding cuts) for the problem. The value 0 (zero), the default, specifies that the attempt to generate MIR cuts should continue only if it seems to be helping.

Value Meaning

-1	Do not generate MIR cuts
0	Automatic: let CPLEX choose; default
1	Generate MIR cuts moderately
2	Generate MIR cuts aggressively

precision of numerical output in MPS and REW file formats

Purpose

Precision of numerical output in MPS and REW file formats

Syntax

C Name	CPX_PARAM_MPSLONGNUM (int)
C++ Name	MPSLongNum (bool)
Java Name	MPSLongNum (bool)
.NET Name	MPSLongNum (bool)
Interactive Optimizer	output mpslong
Identifier	1081

Description

Decides the precision of numerical output in the MPS and REW file formats. When this parameter is set to its default value 1 (one), numbers are written to MPS files in full-precision; that is, up to 15 significant digits may be written. The setting 0 (zero) writes files that correspond to the standard MPS format, where at most 12 characters can be used to represent a value. This limit may result in loss of precision.

Value	bool	Symbol	Meaning
-------	------	--------	---------

0	false	CPX_OFF	Off: use limited MPS precision
1	true	CPX_ON	On: use full-precision; default

See also

MPS file format: industry standard

network logging display switch

Purpose

Network logging display switch

Syntax

C Name	CPX_PARAM_NETDISPLAY (int)
C++ Name	NetDisplay (int)
Java Name	NetDisplay (int)
.NET Name	NetDisplay (int)
OPL Name	netdisplay
Interactive Optimizer	network display
Identifier	5005

Description

Decides what CPLEX reports to the screen during network optimization. Settings 1 and 2 differ only during Phase I. Setting 2 shows monotonic values, whereas 1 usually does not.

Value	Symbol	Meaning
0	CPXNET_NO_DISPLAY_OBJECTIVE	No display
1	CPXNET_TRUE_OBJECTIVE	Display true objective values
2	CPXNET_PENALIZE_OBJECTIVE	Display penalized objective values; default

network optimality tolerance

Purpose

Optimality tolerance for network optimization

Syntax

C Name	CPX_PARAM_NETEPOPT (double)
C++ Name	NetEpOpt (double)
Java Name	NetEpOpt (double)
.NET Name	NetEpOpt (double)
OPL Name	netepopt
Interactive Optimizer	network tolerances optimality
Identifier	5002

Description

Specifies the optimality tolerance for network optimization; that is, the amount a reduced cost may violate the criterion for an optimal solution.

Values

Any number from 1e-11 to 1e-1; **default:** 1e-6.

network primal feasibility tolerance

Purpose

Feasibility tolerance for network primal optimization

Syntax

C Name	CPX_PARAM_NETEPRHS (double)
C++ Name	NetEpRHS (double)
Java Name	NetEpRHS (double)
.NET Name	NetEpRHS (double)
OPL Name	neteprhs
Interactive Optimizer	network tolerances feasibility
Identifier	5003

Description

Specifies feasibility tolerance for network primal optimization. The feasibility tolerance specifies the degree to which the flow value of a model may violate its bounds. This tolerance influences the selection of an optimal basis and can be reset to a higher value when a problem is having difficulty maintaining feasibility during optimization. You may also wish to lower this tolerance after finding an optimal solution if there is any doubt that the solution is truly optimal. If the feasibility tolerance is set too low, CPLEX may falsely conclude that a problem is infeasible. If you encounter reports of infeasibility during Phase II of the optimization, a small adjustment in the feasibility tolerance may improve performance.

Values

Any number from 1e-11 to 1e-1; **default:** 1e-6.

simplex network extraction level

Purpose

Simplex network extraction level

Syntax

C Name	CPX_PARAM_NETFIND (int)
C++ Name	NetFind (int)
Java Name	NetFind (int)
.NET Name	NetFind (int)
OPL Name	netfind
Interactive Optimizer	network netfind
Identifier	1022

Description

Establishes the level of network extraction for network simplex optimization. The default value is suitable for recognizing commonly used modeling approaches when representing a network problem within an LP formulation.

Values

Value	Symbol	Meaning
1	CPX_NETFIND_PURE	Extract pure network only
2	CPX_NETFIND_REFLECT	Try reflection scaling; default
3	CPX_NETFIND_SCALE	Try general scaling

network simplex iteration limit

Purpose

Network simplex iteration limit

Syntax

C Name	CPX_PARAM_NETITLIM (int)
C++ Name	NetItLim (int)
Java Name	NetItLim (int)
.NET Name	NetItLim (int)
OPL Name	netitlim
Interactive Optimizer	network iterations
Identifier	5001

Description

Sets the maximum number of iterations to be performed before the algorithm terminates without reaching optimality.

Values

Any nonnegative integer; **default:** 2100000000.

network simplex pricing algorithm

Purpose

Network simplex pricing algorithm

Syntax

C Name	CPX_PARAM_NETPPRIIND (int)
C++ Name	NetPPriInd (int)
Java Name	NetPPriInd (int)
.NET Name	NetPPriInd (int)
OPL Name	netppriind
Interactive Optimizer	network pricing
Identifier	5004

Description

Specifies the pricing algorithm for network simplex optimization. The default (0) shows best performance for most problems, and currently is equivalent to 3.

Values

Value	Symbol	Meaning
0	CPXNET_PRICE_AUTO	Automatic: let CPLEX choose; default
1	CPXNET_PRICE_PARTIAL	Partial pricing
2	CPXNET_PRICE_MULT_PART	Multiple partial pricing
3	CPXNET_PRICE_SORT_MULT_PART	Multiple partial pricing with sorting

MIP subproblem algorithm

Purpose

MIP subproblem algorithm

Syntax

C Name	CPX_PARAM_SUBALG (int)
C++ Name	NodeAlg (int)
Java Name	NodeAlg (int)
.NET Name	NodeAlg (int)
OPL Name	nodealg
Interactive Optimizer	mip strategy subalgorithm
Identifier	2026

Description

Decides which continuous optimizer will be used to solve the subproblems in a MIP, after the initial relaxation.

The default Automatic setting (0 zero) of this parameter currently selects the dual simplex optimizer for subproblem solution for MILP and MIQP. The Automatic setting may be expanded in the future so that CPLEX chooses the algorithm based on additional characteristics of the model.

For MILP (integer constraints and otherwise continuous variable), all settings are permitted.

For MIQP (integer constraints and positive semi-definite quadratic terms in objective), setting 3 (Network) is not permitted, and setting 5 (Sifting) reverts to 0 (Automatic).

For MIQCP (integer constraints and positive semi-definite quadratic terms among the constraints), only the Barrier optimizer is implemented, and therefore no settings other than 0 (Automatic) and 4 (Barrier) are permitted.

Values

Value	Symbol	Meaning
0	CPX_ALG_AUTOMATIC	Automatic: let CPLEX choose; default
1	CPX_ALG_PRIMAL	Primal simplex
2	CPX_ALG_DUAL	Dual simplex

Value	Symbol	Meaning
3	CPX_ALG_NET	Network simplex
4	CPX_ALG_BARRIER	Barrier
5	CPX_ALG_SIFTING	Sifting

node storage file switch

Purpose

Node storage file switch

Syntax

C Name	CPX_PARAM_NODEFILEIND (int)
C++ Name	NodeFileInd (int)
Java Name	NodeFileInd (int)
.NET Name	NodeFileInd (int)
OPL Name	nodefileind
Interactive Optimizer	mip strategy file
Identifier	2016

Description

Used when working memory (CPX_PARAM_WORKMEM, WorkMem) has been exceeded by the size of the tree. If the node file parameter is set to zero when the tree memory limit is reached, optimization is terminated. Otherwise, a group of nodes is removed from the in-memory set as needed. By default, CPLEX transfers nodes to node files when the in-memory set is larger than 128 MBytes, and it keeps the resulting node files in compressed form in memory. At settings 2 and 3, the node files are transferred to disk, in uncompressed and compressed form respectively, into a directory named by the working directory parameter (CPX_PARAM_WORKDIR, WorkDir), and CPLEX actively manages which nodes remain in memory for processing.

Value Meaning

0	No node file
1	Node file in memory and compressed; default
2	Node file on disk
3	Node file on disk and compressed

See also

directory for working files
memory available for working storage

MIP node limit

Purpose

MIP node limit

Syntax

C Name	CPX_PARAM_NODELIM (int)
C++ Name	NodeLim (int)
Java Name	NodeLim (int)
.NET Name	NodeLim (int)
OPL Name	nodelim
Interactive Optimizer	mip limits nodes
Identifier	2017

Description

Sets the maximum number of nodes solved before the algorithm terminates without reaching optimality. When this parameter is set to 0 (zero), CPLEX completes processing at the root; that is, it creates cuts and applies heuristics at the root. When this parameter is set to 1 (one), it allows branching from the root; that is, nodes are created but not solved.

Values

Any nonnegative integer; **default:** 2100000000.

MIP node selection strategy

Purpose

MIP node selection strategy

Syntax

C Name	CPX_PARAM_NODESEL (int)
C++ Name	NodeSel (int)
Java Name	NodeSel (int)
.NET Name	NodeSel (int)
OPL Name	nodesel
Interactive Optimizer	mip strategy nodeselect
Identifier	2018

Description

Used to set the rule for selecting the next node to process when backtracking. The depth-first search strategy chooses the most recently created node. The best-bound strategy chooses the node with the best objective function for the associated LP relaxation. The best-estimate strategy selects the node with the best estimate of the integer objective value that would be obtained from a node once all integer infeasibilities are removed. An alternative best-estimate search is also available.

Values

Value	Symbol	Meaning
0	CPX_NODESEL_DFS	Depth-first search
1	CPX_NODESEL_BESTBOUND	Best-bound search; default
2	CPX_NODESEL_BESTEST	Best-estimate search
3	CPX_NODESEL_BESTEST_ALT	Alternative best-estimate search

numerical precision emphasis

Purpose

Numerical precision emphasis

Syntax

C Name	CPX_PARAM_NUMERICALEMPHASIS (int)
C++ Name	NumericalEmphasis (bool)
Java Name	NumericalEmphasis (bool)
.NET Name	NumericalEmphasis (bool)
OPL Name	numericalemphasis
Interactive Optimizer	emphasis numerical
Identifier	1083

Description

Emphasizes precision in numerically unstable or difficult problems. This parameter lets you indicate to CPLEX that it should emphasize precision in numerically difficult or unstable problems, with consequent performance trade-offs in time and memory.

Values

Value	bool	Symbol	Meaning
0	false	CPX_OFF	Do not emphasize numerical precision; default
1	true	CPX_ON	Exercise extreme caution in computation

nonzero element read limit

Purpose

Nonzero element read limit

Syntax

C Name	CPX_PARAM_NZREADLIM (int)
C++ Name	NzReadLim (int)
Java Name	NzReadLim (int)
.NET Name	NzReadLim (int)
Interactive Optimizer	read nonzeros
Identifier	1024

Description

Specifies a limit for the number of nonzero elements to read for an allocation of memory. This parameter does not restrict the size of a problem. Rather, it indirectly specifies the default amount of memory that will be pre-allocated before a problem is read from a file. If the limit is exceeded, more memory is automatically allocated.

Values

Any integer from 0 to 268 435 450; **default:** 250 000.

absolute objective difference cutoff

Purpose

Absolute objective difference cutoff

Syntax

C Name	CPX_PARAM_OBJDIF (double)
C++ Name	ObjDif (double)
Java Name	ObjDif (double)
.NET Name	ObjDif (double)
OPL Name	objdif
Interactive Optimizer	mip tolerances objdifference
Identifier	2019

Description

Used to update the cutoff each time a mixed integer solution is found. This absolute value is subtracted from (added to) the newly found integer objective value when minimizing (maximizing). This forces the mixed integer optimization to ignore integer solutions that are not at least this amount better than the best one found so far.

The objective difference parameter can be adjusted to improve problem solving efficiency by limiting the number of nodes; however, setting this parameter at a value other than zero (the default) can cause some integer solutions, including the true integer optimum, to be missed.

Negative values for this parameter can result in some integer solutions that are worse than or the same as those previously generated, but does not necessarily result in the generation of all possible integer solutions.

Values

Any number; **default:** 0.0.

See also

relative objective difference cutoff

lower objective value limit

Purpose

Lower objective value limit

Syntax

C Name	CPX_PARAM_OBJLLIM (double)
C++ Name	ObjLLim (double)
Java Name	ObjLLim (double)
.NET Name	ObjLLim (double)
OPL Name	objllim
Interactive Optimizer	simplex limits lowerobj
Identifier	1025

Description

Sets a lower limit on the value of the objective function in the simplex algorithms. Setting a lower objective function limit causes CPLEX to halt the optimization process when the minimum objective function value limit has been reached. This limit applies only during Phase II of the simplex algorithm in minimization problems.

Tip: This parameter is not effective with the conflict refiner nor with FeasOpt. That is, neither of those tools can analyze an infeasibility introduced by this parameter. If you want to analyze such a condition, add an explicit objective constraint, such as `obj >= c`, to your model instead before you invoke either of those tools.

Values

Any number; **default:** -1e+75.

upper objective value limit

Purpose

Upper objective value limit

Syntax

C Name	CPX_PARAM_OBJULIM (double)
C++ Name	ObjULim (double)
Java Name	ObjULim (double)
.NET Name	ObjULim (double)
OPL Name	objulim
Interactive Optimizer	simplex limits upperobj
Identifier	1026

Description

Sets an upper limit on the value of the objective function in the simplex algorithms. Setting an upper objective function limit causes CPLEX to halt the optimization process when the maximum objective function value limit has been reached. This limit applies only during Phase II of the simplex algorithm in maximization problems.

Tip: This parameter is not effective with the conflict refiner nor with FeasOpt. That is, neither of those tools can analyze an infeasibility introduced by this parameter. If you want to analyze such a condition, add an explicit objective constraint, such as `obj <= c.` to your model instead before you invoke either of those tools.

Values

Any number; **default:** 1e+75.

parallel mode switch

Purpose

Parallel mode switch

Syntax

C Name	CPX_PARAM_PARALLELMODE (int)
C++ Name	ParallelMode (int)
Java Name	ParallelMode (int)
.NET Name	ParallelMode (int)
OPL Name	parallelmode
Interactive Optimizer	parallel
Identifier	1109

Description

Sets the parallel optimization mode. Possible modes are automatic, deterministic, and opportunistic.

In this context, *deterministic* means that multiple runs with the same model at the same parameter settings on the same platform will reproduce the same solution path and results. In contrast, *opportunistic* implies that even slight differences in timing among threads or in the order in which tasks are executed in different threads may produce a different solution path and consequently different timings or different solution vectors during optimization executed in parallel threads. In multithreaded applications, the opportunistic setting entails less synchronization between threads and consequently may provide better performance.

By default, CPLEX applies as much parallelism as possible while still achieving deterministic results. That is, when you run the same model twice on the same platform with the same parameter settings, you will see the same solution and optimization run. This condition is referred to as the *deterministic* mode.

More opportunities to exploit parallelism are available if you do not require determinism. In other words, CPLEX can find more possibilities for parallelism if you do not require an invariant, repeatable solution path and precisely the same solution vector. To use all available parallelism, you need to select the *opportunistic* parallel mode. In this mode, CPLEX will utilize all opportunities for parallelism in order to achieve best performance.

However, in opportunistic mode, the actual optimization may differ from run to run, including the solution time itself and the path traveled in the search.

Deterministic and sequential optimization

A truly parallel deterministic algorithm is available only for MIP optimization.

Only opportunistic parallel algorithms (barrier and concurrent optimizers) are available for continuous models. (Each of the simplex algorithms runs sequentially on a continuous model.)

Consequently, when parallel mode is set to deterministic, both barrier and concurrent optimizers are restricted to run only sequentially, not in parallel.

Interaction with the threads parameter

Settings of this parallel mode parameter interact with settings of the *global default thread count* parameter (Threads, CPX_PARAM_THREADS) as summarized in the tables:

- ◆ *Interaction of Callbacks with Threads and Parallel Mode Parameters: No Callbacks or only Informational Callbacks in Application*
- ◆ *Interaction of Callbacks with Threads and Parallel Mode Parameters: Only Query Callbacks in Application*
- ◆ *Interaction of Callbacks with Threads and Parallel Mode Parameters: Control Callbacks in Application*

The default (automatic) setting of the parallel mode parameter allows CPLEX to choose between deterministic and opportunistic mode depending on the threads parameter. If the threads parameter is set to its automatic setting (the default), CPLEX chooses deterministic mode.

If the threads parameter is set to one, CPLEX runs sequentially in deterministic mode in a single thread.

Otherwise, if the threads parameter is set to a value greater than one, CPLEX chooses opportunistic mode.

Callbacks and MIP optimization

If callbacks other than informational callbacks are used for solving a MIP, the order in which the callbacks are called cannot be guaranteed to remain deterministic, not even in deterministic mode. Thus, to make sure of deterministic runs when the parallel mode parameter is at its default setting, CPLEX will revert to sequential solving of the MIP in the presence of query callbacks, diagnostic callbacks, or control callbacks.

Consequently, if your application invokes query, diagnostic, or control callbacks, and you still prefer deterministic search, you can choose value 1 (one), overriding the automatic setting and turning on deterministic search. It is then your responsibility to make sure that your callbacks do not perform operations that could lead to opportunistic behavior and are implemented in a thread-safe way. To meet these conditions, your application must **not** store and must **not** update any information in the callbacks.

Determinism vs opportunism

This parameter also allows you to turn off this default setting by choosing value -1 (minus one). Cases where you might wish to turn off deterministic search include situations where you want to take advantage of possibly faster performance of opportunistic parallel MIP optimization in multiple threads after you have confirmed that deterministic parallel MIP optimization produced the results you expected.

Values

Value	Symbolic Constant Callable Library	Symbolic Constant Concert Technology	Meaning
-1	CPX_PARALLEL_OPPORTUNISTIC	Opportunistic	Enable opportunistic parallel search mode
0	CPX_PARALLEL_AUTO	AutoParallel	Automatic: let CPLEX decide whether to invoke deterministic or opportunistic search, depending on the threads parameter; default
1	CPX_PARALLEL_DETERMINISTIC	Deterministic	Enable deterministic parallel search mode

See also: *global default thread count*: CPX_PARAM_THREADS, Threads

simplex perturbation switch

Purpose

Simplex perturbation switch

Syntax

C Name	CPX_PARAM_PERIND (int)
C++ Name	PerInd (bool)
Java Name	PerInd (bool)
.NET Name	PerInd (bool)
OPL Name	perind
Interactive Optimizer	simplex perturbation
Identifier	1027

Description

Decides whether to perturb problems.

Setting this parameter to 1 (one) causes all problems to be automatically perturbed as optimization begins. A setting of 0 (zero) allows CPLEX to decide dynamically, during solution, whether progress is slow enough to merit a perturbation. The situations in which a setting of 1 (one) helps are rare and restricted to problems that exhibit extreme degeneracy.

Values

Value	bool	Symbol	Meaning
0	false	CPX_OFF	Automatic: let CPLEX choose; default
1	true	CPX_ON	Turn on perturbation from beginning

simplex perturbation limit

Purpose

Simplex perturbation limit

Syntax

C Name	CPX_PARAM_PERLIM (int)
C++ Name	PerLim (int)
Java Name	PerLim (int)
.NET Name	PerLim (int)
OPL Name	perlim
Interactive Optimizer	simplex limits perturbation
Identifier	1028

Description

Sets the number of degenerate iterations before perturbation is performed.

Values

Value	Meaning
0	Automatic: let CPLEX choose; default
Any positive integer	Number of degenerate iterations before perturbation

absolute MIP gap before starting to polish a feasible solution

Purpose

Absolute MIP gap before starting to polish a feasible solution

Syntax

C Name	CPX_PARAM_POLISHAFTEREPAGAP (double)
C++ Name	PolishAfterEpAGap (double)
Java Name	PolishAfterEpAGap (double)
.NET Name	PolishAfterEpAGap (double)
OPL Name	
Interactive Optimizer	mip polishafter absmipgap
Identifier	2126

Description

Sets an absolute MIP gap (that is, the difference between the best integer objective and the objective of the best node remaining) after which ILOG CPLEX stops branch-and-cut and begins polishing a feasible solution. The default value (0.0) is such that ILOG CPLEX does not invoke solution polishing by default.

Starting conditions

ILOG CPLEX must have a feasible solution in order to start polishing. It must also have certain internal structures in place to support solution polishing. Consequently, when the criterion specified by this parameter is met, ILOG CPLEX begins solution polishing only after these starting conditions are also met. That is, there may be a delay between the moment when the criterion specified by this parameter is met and when solution polishing starts.

Values

Any nonnegative value; **default:** 0.0.

See also

absolute MIP gap tolerance

relative MIP gap before starting to polish a feasible solution

Purpose

Relative MIP gap before starting to polish a solution

Syntax

C Name	CPX_PARAM_POLISHAFTEREPGAP (double)
C++ Name	PolishAfterEpGap (double)
Java Name	PolishAfterEpGap (double)
.NET Name	PolishAfterEpGap (double)
OPL Name	
Interactive Optimizer	mip polishafter mipgap
Identifier	2127

Description

Sets a relative MIP gap after which ILOG CPLEX will stop branch-and-cut and begin polishing a feasible solution. The default value (0.0) is such that ILOG CPLEX does not invoke solution polishing by default. The relative MIP gap is calculated like this:

$$\frac{|\text{bestnode-bestinteger}|}{(1e-10+|\text{bestinteger}|)}$$

Starting conditions

ILOG CPLEX must have a feasible solution in order to start polishing. It must also have certain internal structures in place to support solution polishing. Consequently, when the criterion specified by this parameter is met, ILOG CPLEX begins solution polishing only after these starting conditions are also met. That is, there may be a delay between the moment when the criterion specified by this parameter is met and when solution polishing starts.

Values

Any number from 0.0 to 1.0, inclusive; **default**: 0.0.

See also

relative MIP gap tolerance

MIP integer solutions to find before starting to polish a feasible solution

Purpose

MIP integer solutions to find before starting to polish a feasible solution

Syntax

C Name	CPX_PARAM_POLISHAFTERINTSOL (int)
C++ Name	PolishAfterIntSol (int)
Java Name	PolishAfterIntSol (int)
.NET Name	PolishAfterIntSol (int)
OPL Name	
Interactive Optimizer	mip polishafter solutions
Identifier	2129

Description

Sets the number of integer solutions to find before CPLEX stops branch-and-cut and begins to polish a feasible solution. The default value is such that ILOG CPLEX does not invoke solution polishing by default.

Starting conditions

ILOG CPLEX must have a feasible solution in order to start polishing. It must also have certain internal structures in place to support solution polishing. Consequently, when the criterion specified by this parameter is met, ILOG CPLEX begins solution polishing only after these starting conditions are also met. That is, there may be a delay between the moment when the criterion specified by this parameter is met and when solution polishing starts.

Values

Any positive integer strictly greater than zero; zero is **not** allowed; **default**: 2 100 000 000

See also

MIP integer solution limit

nodes to process before starting to polish a feasible solution

Purpose

Nodes to process before starting to polish a feasible solution

Syntax

C Name	CPX_PARAM_POLISHAFTERNODE (int)
C++ Name	PolishAfterNode (int)
Java Name	PolishAfterNode (int)
.NET Name	PolishAfterNode (int)
OPL Name	
Interactive Optimizer	mip polishafter nodes
Identifier	2128

Description

Sets the number of nodes processed in branch-and-cut before ILOG CPLEX starts solution polishing, if a feasible solution is available.

When this parameter is set to 0 (zero), ILOG CPLEX completes processing at the root; that is, it creates cuts and applies heuristics at the root.

When this parameter is set to 1 (one), it allows branching from the root; that is, nodes are created but not solved.

When no feasible solution is available yet, ILOG CPLEX explores more nodes than the number specified by this parameter.

Starting conditions

ILOG CPLEX must have a feasible solution in order to start polishing. It must also have certain internal structures in place to support solution polishing. Consequently, when the criterion specified by this parameter is met, ILOG CPLEX begins solution polishing only after these starting conditions are also met. That is, there may be a delay between the moment when the criterion specified by this parameter is met and when solution polishing starts.

Values

Any nonnegative integer; **default:** 2 100 000 000

See also

MIP node limit

time before starting to polish a feasible solution

Purpose

Time before starting to polish a feasible solution

Syntax

C Name	CPX_PARAM_POLISHAFTERTIME (double)
C++ Name	PolishAfterTime (double)
Java Name	PolishAfterTime (double)
.NET Name	PolishAfterTime (double)
OPL Name	
Interactive Optimizer	mip polishafter time
Identifier	2130

Description

Tells CPLEX how much time in seconds to spend during mixed integer optimization before CPLEX starts polishing a feasible solution. The default value (1.0E+75 seconds) is such that ILOG CPLEX does **not** start solution polishing by default.

Whether CPLEX measures CPU time or wall clock time (also known as real time) depends on the parameter *clock type for computation time*.

Starting conditions

ILOG CPLEX must have a feasible solution in order to start polishing. It must also have certain internal structures in place to support solution polishing. Consequently, when the criterion specified by this parameter is met, ILOG CPLEX begins solution polishing only after these starting conditions are also met. That is, there may be a delay between the moment when the criterion specified by this parameter is met and when solution polishing starts.

Values

Any nonnegative value in seconds; **default**:1.0E+75 seconds.

See also

clock type for computation time

time spent polishing a solution (deprecated)

Purpose

Time spent polishing a solution (deprecated)

Syntax

C Name	CPX_PARAM_POLISHTIME (double)
C++ Name	PolishTime (double)
Java Name	PolishTime (double)
.NET Name	PolishTime (double)
OPL Name	polishtime
Interactive Optimizer	mip limit polishtime
Identifier	2066

Description

This **deprecated** parameter told CPLEX how much time in seconds to spend after a normal mixed integer optimization in polishing a solution. The default was zero, no polishing time.

Instead of this **deprecated** parameter, use one of the following parameters to control the effort that CPLEX spends in branch-and-cut before it begins polishing a feasible solution:

- ◆ *absolute MIP gap before starting to polish a feasible solution*
- ◆ *relative MIP gap before starting to polish a feasible solution*
- ◆ *MIP integer solutions to find before starting to polish a feasible solution*
- ◆ *nodes to process before starting to polish a feasible solution*
- ◆ *time before starting to polish a feasible solution*
- ◆ *optimizer time limit*

Values

Any nonnegative value in seconds; **default**: 0.0 (zero) seconds.

limit on number of solutions generated for solution pool

Purpose

Limit on number of solutions generated for the solution pool

Syntax

C Name	CPX_PARAM_POPULATELIM (int)
C++ Name	PopulateLim (int)
Java Name	PopulateLim (int)
.NET Name	PopulateLim (int)
OPL Name	populatelim
Interactive Optimizer	mip limits populate
Identifier	2108

Description

Limits the number of mixed integer programming (MIP) solutions generated for the solution pool during each call to the populate procedure. Populate stops when it has generated `PopulateLim` solutions. A solution is counted if it is valid for all filters, consistent with the relative and absolute pool gap parameters, and has not been rejected by the incumbent callback (if any exists), whether or not it improves the objective of the model.

In parallel, populate may not respect this parameter exactly due to disparities between threads. That is, it may happen that populate stops when it has generated a number of solutions slightly more than or slightly less than this limit because of differences in synchronization between threads.

This parameter does **not** apply to MIP optimization generally; it applies only to the populate procedure.

If you are looking for a parameter to control the number of solutions stored in the solution pool, consider instead the solution pool capacity parameter (*limit on number of solutions kept in solution pool*: `SolnPoolCapacity`, `CPX_PARAM_SOLNPOOLCAPACITY`).

Populate will stop before it reaches the limit set by this parameter if it reaches another limit, such as a time limit set by the user. Additional stopping criteria can be specified by these parameters:

- ◆ *relative gap for solution pool*: `SolnPoolGap`, `CPX_PARAM_SOLNPOOLGAP`

- ◆ *absolute gap for solution pool*: SolnPoolAGap, CPX_PARAM_SOLNPOOLAGAP
- ◆ *MIP node limit*: NodeLim, CPX_PARAM_NODELIM
- ◆ *optimizer time limit*: TiLim, CPX_PARAM_TILIM

Values

Any nonnegative integer; **default**: 20.

primal simplex pricing algorithm

Purpose

Primal simplex pricing algorithm

Syntax

C Name	CPX_PARAM_PPRIIND (int)
C++ Name	PPriInd (int)
Java Name	PPriInd (int)
.NET Name	PPriInd (int)
OPL Name	ppriind
Interactive Optimizer	simplex pgradient
Identifier	1029

Description

Sets the primal simplex pricing algorithm. The default pricing (0) usually provides the fastest solution time, but many problems benefit from alternative settings.

Values

Value	Symbol	Meaning
-1	CPX_PPRIIND_PARTIAL	Reduced-cost pricing
0	CPX_PPRIIND_AUTO	Hybrid reduced-cost & devex pricing; default
1	CPX_PPRIIND_DEVEX	Devex pricing
2	CPX_PPRIIND_STEEP	Steepest-edge pricing
3	CPX_PPRIIND_STEEPQSTART	Steepest-edge pricing with slack initial norms
4	CPX_PPRIIND_FULL	Full pricing

presolve dual setting

Purpose

Presolve dual setting

Syntax

C Name	CPX_PARAM_PREDUAL (int)
C++ Name	PreDual (int)
Java Name	PreDual (int)
.NET Name	PreDual (int)
OPL Name	predual
Interactive Optimizer	preprocessing dual
Identifier	1044

Description

Decides whether CPLEX presolve should pass the primal or dual linear programming problem to the linear programming optimization algorithm. By default, CPLEX chooses automatically.

If this parameter is set to 1 (one), the CPLEX presolve algorithm is applied to the primal problem, but the resulting dual linear program is passed to the optimizer. This is a useful technique for problems with more constraints than variables.

Values

Value	Meaning
-1	Turn off this feature
0	Automatic: let CPLEX choose; default
1	Turn on this feature

presolve switch

Purpose

Presolve switch

Syntax

C Name	CPX_PARAM_PREIND (int)
C++ Name	PreInd (bool)
Java Name	PreInd (bool)
.NET Name	PreInd (bool)
OPL Name	preind
Interactive Optimizer	preprocessing presolve
Identifier	1030

Description

Decides whether CPLEX applies presolve during preprocessing. When set to 1 (one), the default, this parameter invokes the CPLEX presolve to simplify and reduce problems.

Values

Value	bool	Symbol	Meaning
0	false	CPX_OFF	Do not apply presolve
1	true	CPX_ON	Apply presolve; default

linear reduction switch

Purpose

Linear reduction switch

Syntax

C Name	CPX_PARAM_PRELINEAR (int)
C++ Name	PreLinear (int)
Java Name	PreLinear (int)
.NET Name	PreLinear (int)
OPL Name	prelinear
Interactive Optimizer	preprocessing linear
Identifier	1058

Description

Decides whether linear or full reductions occur during preprocessing. If only linear reductions are performed, each variable in the original model can be expressed as a linear form of variables in the presolved model. This condition guarantees, for example, that users can add their own custom cuts to the presolved model.

Values

Value	Meaning
0	Perform only linear reductions
1	Perform full reductions; default

limit on the number of presolve passes made

Purpose

Limit on the number of presolve passes made

Syntax

C Name	CPX_PARAM_PREPASS (int)
C++ Name	PrePass (int)
Java Name	PrePass (int)
.NET Name	PrePass (int)
OPL Name	prepass
Interactive Optimizer	preprocessing numpass
Identifier	1052

Description

Limits the number of presolve passes that CPLEX makes during preprocessing. When this parameter is set to a nonzero value, invokes CPLEX presolve to simplify and reduce problems.

When this parameter is set to a positive value, presolve is applied the specified number of times, or until no more reductions are possible.

At the default value of -1, presolve should continue only if it seems to be helping.

When this parameter is set to zero, CPLEX does not apply presolve, but other reductions may occur, depending on settings of other parameters and specifics of your model.

Values

Value	Meaning
-1	Automatic: let CPLEX choose; presolve continues as long as helpful; default
0	Do not use presolve; other reductions may still occur
Any positive integer	Apply presolve specified number of times

node presolve switch

Purpose

Node presolve switch

Syntax

C Name	CPX_PARAM_PRESLVND (int)
C++ Name	PreslvNd (int)
Java Name	PreslvNd (int)
.NET Name	PreslvNd (int)
OPL Name	preslvnd
Interactive Optimizer	mip strategy presolvenode
Identifier	2037

Description

Decides whether node presolve should be performed at the nodes of a mixed integer programming (MIP) solution. Node presolve can significantly reduce solution time for some models. The default setting is generally effective at deciding whether to apply node presolve, although runtimes can be reduced for some models by the user turning node presolve off.

Value Meaning

-1	No node presolve
0	Automatic: let CPLEX choose; default
1	Force presolve at nodes
2	Perform probing on integer-infeasible variables

simplex pricing candidate list size

Purpose

Simplex pricing candidate list size

Syntax

C Name	CPX_PARAM_PRICELIM (int)
C++ Name	PriceLim (int)
Java Name	PriceLim (int)
.NET Name	PriceLim (int)
OPL Name	pricelim
Interactive Optimizer	simplex pricing
Identifier	1010

Description

Sets the maximum number of variables kept in the list of pricing candidates for the simplex algorithms.

Values

Value	Meaning
0	Automatic: let CPLEX choose; default
Any positive integer	Number of pricing candidates

MIP probing level

Purpose

MIP probing level

Syntax

C Name	CPX_PARAM_PROBE (int)
C++ Name	Probe (int)
Java Name	Probe (int)
.NET Name	Probe (int)
OPL Name	probe
Interactive Optimizer	mip strategy probe
Identifier	2042

Description

Sets the amount of probing on variables to be performed before MIP branching. Higher settings perform more probing. Probing can be very powerful but very time-consuming at the start. Setting the parameter to values above the default of 0 (automatic) can result in dramatic reductions or dramatic increases in solution time, depending on the model.

Values

Value	Meaning
-1	No probing
0	Automatic: let CPLEX choose; default
1	Moderate probing level
2	Aggressive probing level
3	Very aggressive probing level

time spent probing

Purpose

Time spent probing

Syntax

C Name	CPX_PARAM_PROBETIME (double)
C++ Name	ProbeTime (double)
Java Name	ProbeTime (double)
.NET Name	ProbeTime (double)
OPL Name	probetime
Interactive Optimizer	mip limit probetime
Identifier	2065

Description

Limits the amount of time in seconds spent probing.

Values

Any nonnegative number; **default:** 1e+75.

indefinite MIQP switch

Purpose

Indefinite MIQP switch

Syntax

C Name	CPX_PARAM_QPMAKEPSDIND (int)
C++ Name	QPmakePSDInd (bool)
Java Name	QPmakePSDInd (bool)
.NET Name	QPmakePSDInd (bool)
OPL Name	qpmakepsdind
Interactive Optimizer	preprocessing qpmakepsd
Identifier	4010

Description

Decides whether CPLEX will attempt to reformulate a MIQP or MIQCP model that contains only binary variables. When this feature is active, adjustments will be made to the elements of a quadratic matrix that is not nominally positive semi-definite (PSD, as required by CPLEX for all QP and most QCP formulations), to make it PSD, and CPLEX will also attempt to tighten an already PSD matrix for better numerical behavior. The default setting of 1 (one) means yes, CPLEX should attempt to reformulate, but you can turn it off if necessary; most models should benefit from the default setting.

Values

Value	bool	Symbol	Meaning
0	false	CPX_OFF	Turn off attempts to make binary model PSD
1	true	CPX_ON	On: CPLEX attempts to make binary model PSD; default

QP Q-matrix nonzero read limit

Purpose

QP Q matrix nonzero read limit

Syntax

C Name	CPX_PARAM_QPNZREADLIM (int)
C++ Name	QPNzReadLim (int)
Java Name	QPNzReadLim (int)
.NET Name	QPNzReadLim (int)
Interactive Optimizer	read qpnonzeros
Identifier	4001

Description

Specifies a limit for the number of nonzero elements to read for an allocation of memory in a model with a quadratic matrix.

This parameter does not restrict the size of a problem. Rather, it indirectly specifies the default amount of memory that will be pre-allocated before a problem is read from a file. If the limit is exceeded, more memory is automatically allocated.

Values

Any integer from 0 to 268 435 450; **default:** 5 000.

primal and dual reduction type

Purpose

Primal and dual reduction type

Syntax

C Name	CPX_PARAM_REDUCE (int)
C++ Name	Reduce (int)
Java Name	Reduce (int)
.NET Name	Reduce (int)
OPL Name	reduce
Interactive Optimizer	preprocessing reduce
Identifier	1057

Description

Decides whether primal reductions, dual reductions, both, or neither are performed during preprocessing.

Values

Value	Symbol	Meaning
0	CPX_PREREDUCE_NOPRIMALORDUAL	No primal or dual reductions
1	CPX_PREREDUCE_PRIMALONLY	Only primal reductions
2	CPX_PREREDUCE_DUALONLY	Only dual reductions
3	CPX_PREREDUCE_PRIMALANDDUAL	Both primal and dual reductions; default

simplex refactoring frequency

Purpose

Simplex refactoring frequency

Syntax

C Name	CPX_PARAM_REINV (int)
C++ Name	ReInv (int)
Java Name	ReInv (int)
.NET Name	ReInv (int)
OPL Name	reinv
Interactive Optimizer	simplex refactor
Identifier	1031

Description

Sets the number of iterations between refactoring of the basis matrix.

Values

Value	Meaning
0	Automatic: let CPLEX choose; default
Integer from 1 to 10 000	Number of iterations between refactoring of the basis matrix

relaxed LP presolve switch

Purpose

Relaxed LP presolve switch

Syntax

C Name	CPX_PARAM_RELAXPREIND (int)
C++ Name	RelaxPreInd (int)
Java Name	RelaxPreInd (int)
.NET Name	RelaxPreInd (int)
OPL Name	relaxpreind
Interactive Optimizer	preprocessing relax
Identifier	2034

Description

Decides whether LP presolve is applied to the root relaxation in a mixed integer program (MIP). Sometimes additional reductions can be made beyond any MIP presolve reductions that were already done. By default, CPLEX applies presolve to the initial relaxation in order to hasten time to the initial solution.

Value	Symbol	Meaning
-------	--------	---------

-1		Automatic: let CPLEX choose; default
0	CPX_OFF	Off: do not use presolve on initial relaxation
1	CPX_ON	On: use presolve on initial relaxation

relative objective difference cutoff

Purpose

Relative objective difference cutoff

Syntax

C Name	CPX_PARAM_RELOBJDIF (double)
C++ Name	RelObjDif (double)
Java Name	RelObjDif (double)
.NET Name	RelObjDif (double)
OPL Name	relobjdif
Interactive Optimizer	mip tolerances relobjdifference
Identifier	2022

Description

Used to update the cutoff each time a mixed integer solution is found. The value is multiplied by the absolute value of the integer objective and subtracted from (added to) the newly found integer objective when minimizing (maximizing). This computation forces the mixed integer optimization to ignore integer solutions that are not at least this amount better than the one found so far.

The relative objective difference parameter can be adjusted to improve problem solving efficiency by limiting the number of nodes; however, setting this parameter at a value other than zero (the default) can cause some integer solutions, including the true integer optimum, to be missed.

If both the relative objective difference and the *absolute objective difference cutoff* (CPX_PARAM_OBJDIF, ObjDif) are nonzero, the value of the absolute objective difference is used.

Values

Any number from 0.0 to 1.0; **default:** 0.0.

See also

absolute objective difference cutoff

frequency to try to repair infeasible MIP start

Purpose

Frequency to try to repair infeasible MIP start

Syntax

C Name	CPX_PARAM_REPAIRTRIES (int)
C++ Name	RepairTries (int)
Java Name	RepairTries (int)
.NET Name	RepairTries (int)
OPL Name	repairtries
Interactive Optimizer	mip limits repairtries
Identifier	2067

Description

Limits the attempts to repair an infeasible MIP start. This parameter lets you tell CPLEX whether and how many times it should try to repair an infeasible MIP start that you supplied. The parameter has no effect if the MIP start you supplied is feasible. It has no effect if no MIP start was supplied.

Values

Value	Meaning
-1	None: do not try to repair
0	Automatic: let CPLEX choose; default
Any positive integer	Frequency to attempt repairs

MIP repeat presolve switch

Purpose

Reapply presolve after processing the root node

Syntax

C Name	CPX_PARAM_REPEATPRESOLVE (int)
C++ Name	RepeatPresolve (int)
Java Name	RepeatPresolve (int)
.NET Name	RepeatPresolve (int)
OPL Name	repeatpresolve
Interactive Optimizer	preprocessing repeatpresolve
Identifier	2064

Description

Decides whether to re-apply presolve, with or without cuts, to a MIP model after processing at the root is otherwise complete.

Values

Value	Symbol
-------	--------

-1	Automatic: let CPLEX choose; default
0	Turn off represolve
1	Represolve without cuts
2	Represolve with cuts
3	Represovle with cuts and allow new root cuts

RINS heuristic frequency

Purpose

RINS heuristic frequency

Syntax

C Name	CPX_PARAM_RINSHEUR (int)
C++ Name	RINSHeur (int)
Java Name	RINSHeur (int)
.NET Name	RINSHeur (int)
OPL Name	rinsheur
Interactive Optimizer	mip strategy rinsheur
Identifier	2061

Description

Decides how often to apply the relaxation induced neighborhood search (RINS) heuristic. This heuristic attempts to improve upon the best solution found so far. It will not be applied until CPLEX has found at least one incumbent solution.

Setting the value to -1 turns off the RINS heuristic. Setting the value to 0 (zero), the default, applies the RINS heuristic at an interval chosen automatically by CPLEX. Setting the value to a positive number applies the RINS heuristic at the requested node interval. For example, setting RINSHeur to 20 dictates that the RINS heuristic be called at node 0, 20, 40, 60, etc.

RINS is a powerful heuristic for finding high quality feasible solutions, but it may be expensive.

Values

Value	Meaning
-1	None: do not apply RINS heuristic
0	Automatic: let CPLEX choose; default
Any positive integer	Frequency to apply RINS heuristic

algorithm for continuous problems

Purpose

Solution algorithm for continuous problems

Syntax

C Name	CPX_PARAM_LPMETHOD (int)
C++ Name	RootAlg (int)
Java Name	RootAlg (int)
.NET Name	RootAlg (int)
OPL Name	rootalg
Interactive Optimizer	lpmethod
Identifier	1062

Description

Controls which algorithm is used to solve continuous models or to solve the root relaxation of a MIP. In the object-oriented APIs, you make this selection through the `RootAlg` parameter. In the C API and the Interactive Optimizer, there are separate parameters to control LP, QP, and MIP optimizers, depending on the problem type.

In all cases, the default setting is 0 (zero). The default setting means that CPLEX will select the algorithm in a way that should give best overall performance.

For specific problem classes, the following details document the automatic settings. Note that future versions of CPLEX could adopt different strategies. Therefore, if you select any nondefault settings, you should review them periodically.

Currently, the behavior of the automatic setting is that CPLEX almost always invokes the dual simplex algorithm when it is solving an LP model from scratch. When it is continuing from an advanced basis, it will check whether the basis is primal or dual feasible, and choose the primal or dual simplex algorithm accordingly.

If multiple threads have been requested, the concurrent optimization algorithm is selected by the automatic setting.

The automatic setting may be expanded in the future so that CPLEX chooses the algorithm based on additional problem characteristics.

Values

Value	Symbol	Meaning
0	CPX_ALG_AUTOMATIC	Automatic: let CPLEX choose; default
1	CPX_ALG_PRIMAL	Primal simplex
2	CPX_ALG_DUAL	Dual simplex
3	CPX_ALG_NET	Network simplex
4	CPX_ALG_BARRIER	Barrier
5	CPX_ALG_SIFTING	Sifting
6	CPX_ALG_CONCURRENT	Concurrent (Dual, Barrier, and Primal)

algorithm for continuous quadratic optimization

Purpose

Algorithm for continuous quadratic optimization

Syntax

C Name	CPX_PARAM_QPMETHOD (int)
C++ Name	RootAlg (int)
Java Name	RootAlg (int)
.NET Name	RootAlg (int)
OPL Name	rootalg
Interactive Optimizer	qpmethod
Identifier	1063

Description

Sets which algorithm to use when the C routine CPXqpopt (or the command optimize in the Interactive Optimizer) is invoked.

Currently, the behavior of the Automatic setting is that CPLEX invokes the Barrier Optimizer for continuous QP models. The Automatic setting may be expanded in the future so that CPLEX chooses the algorithm based on additional problem characteristics.

Values

Value	Symbol	Meaning
0	CPX_ALG_AUTOMATIC	Automatic: let CPLEX choose; default
4	CPX_ALG_BARRIER	Barrier

MIP starting algorithm

Purpose

MIP starting algorithm

Syntax

C Name	CPX_PARAM_STARTALG (int)
C++ Name	RootAlg (int)
Java Name	RootAlg (int)
.NET Name	RootAlg (int)
OPL Name	rootalg
Interactive Optimizer	mip strategy startalgorithm
Identifier	2025

Description

Sets which continuous optimizer will be used to solve the initial relaxation of a MIP.

The default Automatic setting (0 zero) of this parameter currently selects the dual simplex optimizer for root relaxations for MILP and MIQP. The Automatic setting may be expanded in the future so that CPLEX chooses the algorithm based on additional characteristics of the model.

For MILP (integer constraints and otherwise continuous variables), all settings are permitted.

For MIQP (integer constraints and positive semi-definite quadratic terms in the objective), settings 5 (Sifting) and 6 (Concurrent) are **not** implemented; if you happen to choose them, setting 5 (Sifting) reverts to 0 (ero) and setting 6 (Concurrent) reverts to 4.

For MIQCP (integer constraints and positive semi-definite quadratic terms among the constraints), only the Barrier Optimizer is implemented, and therefore no settings other than 0 (zero) and 4 are permitted.

Values

Value	Symbol	Meaning
0	CPX_ALG_AUTOMATIC	Automatic: let CPLEX choose; default
1	CPX_ALG_PRIMAL	Primal Simplex
2	CPX_ALG_DUAL	Dual Simplex

Value	Symbol	Meaning
3	CPX_ALG_NET	Network Simplex
4	CPX_ALG_BARRIER	Barrier
5	CPX_ALG_SIFTING	Sifting
6	CPX_ALG_CONCURRENT	Concurrent (Dual, Barrier, and Primal)

constraint (row) read limit

Purpose

Constraint (row) read limit

Syntax

C Name	CPX_PARAM_ROWREADLIM (int)
C++ Name	RowReadLim (int)
Java Name	RowReadLim (int)
.NET Name	RowReadLim (int)
Interactive Optimizer	read constraints
Identifier	1021

Description

Specifies a limit for the number of rows (constraints) to read for an allocation of memory.

This parameter does not restrict the size of a problem. Rather, it indirectly specifies the default amount of memory that will be pre-allocated before a problem is read from a file. If the limit is exceeded, more memory is automatically allocated.

Values

Any integer from 0 to 268 435 450; **default:** 30 000.

scale parameter

Purpose

Scale parameter

Syntax

C Name	CPX_PARAM_SCAIND (int)
C++ Name	ScaInd (int)
Java Name	ScaInd (int)
.NET Name	ScaInd (int)
OPL Name	scaind
Interactive Optimizer	read scale
Identifier	1034

Description

Decides how to scale the problem matrix.

Values

Value	Meaning
-1	No scaling
0	Equilibration scaling; default
1	More aggressive scaling

messages to screen switch

Purpose

Messages to screen switch

Syntax

C Name	CPX_PARAM_SCRIND (int)
C++ Name	screen indicator not available in this API
Java Name	screen indicator not available in this API
.NET Name	screen indicator not available in this API
Interactive Optimizer	screen indicator not available in this interface
Identifier	1035

Description

Decides whether or not results are displayed on screen in an application of the C API.

To turn off output to the screen, in a C++ application, where `cplex` is an instance of the class `IloCplex` and `env` is an instance of the class `IloEnv`, the environment, use `cplex.setOut(env.getNullStream())`.

In a Java application, use `cplex.setOut(null)`.

In a .NET application, use `Cplex.SetOut(Null)`.

Values

Value	Symbol	Meaning
0	CPX_OFF	Turn off display of messages to screen; default
1	CPX_ON	Display messages on screen

sifting subproblem algorithm

Purpose

Sifting subproblem algorithm

Syntax

C Name	CPX_PARAM_SIFTALG (int)
C++ Name	SiftAlg (int)
Java Name	SiftAlg (int)
.NET Name	SiftAlg (int)
OPL Name	siftalg
Interactive Optimizer	sifting algorithm
Identifier	1077

Description

Sets the algorithm to be used for solving sifting subproblems. The default automatic setting will typically use a mix of barrier and primal simplex.

Values

Value	Symbol	Meaning
0	CPX_ALG_AUTOMATIC	Automatic: let CPLEX choose; default
1	CPX_ALG_PRIMAL	Primal Simplex
2	CPX_ALG_DUAL	Dual Simplex
3	CPX_ALG_NET	Network Simplex
4	CPX_ALG_BARRIER	Barrier

sifting information display

Purpose

Sifting information display

Syntax

C Name	CPX_PARAM_SIFTDISPLAY (int)
C++ Name	SiftDisplay (int)
Java Name	SiftDisplay (int)
.NET Name	SiftDisplay (int)
OPL Name	siftdisplay
Interactive Optimizer	sifting display
Identifier	1076

Description

Sets the amount of information to display about the progress of sifting.

Values

Value	Meaning
0	No display of sifting information
1	Display major iterations; default
2	Display LP subproblem information within each sifting iteration

upper limit on sifting iterations

Purpose

Upper limit on sifting iterations

Syntax

C Name	CPX_PARAM_SIFTITLIM (int)
C++ Name	SiftItLim (int)
Java Name	SiftItLim (int)
.NET Name	SiftItLim (int)
OPL Name	siftitlim
Interactive Optimizer	sifting iterations
Identifier	1078

Description

Sets the maximum number of sifting iterations that may be performed if convergence to optimality has not been reached.

Values

Any nonnegative integer; **default:** 210000000.

simplex iteration information display

Purpose

Simplex iteration information display

Syntax

C Name	CPX_PARAM_SIMDISPLAY (int)
C++ Name	SimDisplay (int)
Java Name	SimDisplay (int)
.NET Name	SimDisplay (int)
OPL Name	simdisplay
Interactive Optimizer	simplex display
Identifier	1019

Description

Sets how often CPLEX reports about iterations during simplex optimization.

Values

Value	Meaning
0	No iteration messages until solution
1	Iteration information after each refactoring; default
2	Iteration information for each iteration

simplex singularity repair limit

Purpose

Simplex singularity repair limit

Syntax

C Name	CPX_PARAM_SINGLIM (int)
C++ Name	SingLim (int)
Java Name	SingLim (int)
.NET Name	SingLim (int)
OPL Name	singlim
Interactive Optimizer	simplex limits singularity
Identifier	1037

Description

Restricts the number of times CPLEX attempts to repair the basis when singularities are encountered during the simplex algorithm. When this limit is exceeded, CPLEX replaces the current basis with the best factorable basis that has been found.

Values

Any nonnegative integer; **default:** 10.

absolute gap for solution pool

Purpose

Absolute gap for solution pool

Syntax

C Name	CPX_PARAM_SOLNPOOLAGAP (double)
C++ Name	SolnPoolAGap (double)
Java Name	SolnPoolAGap (double)
.NET Name	SolnPoolAGap (double)
OPL Name	solnpoolagap
Interactive Optimizer	mip pool absgap
Identifier	2106

Description

Sets an absolute tolerance on the objective value for the solutions in the solution pool. Solutions that are worse (either greater in the case of a minimization, or less in the case of a maximization) than the objective of the incumbent solution according to this measure are not kept in the solution pool.

Values of the solution pool *absolute gap* (SolnPoolAGap or CPX_PARAM_SOLNPOOLAGAP) and the solution pool *relative gap* (*relative gap for solution pool*: SolnPoolGap or CPX_PARAM_SOLNPOOLGAP) may differ: For example, you may specify that solutions must be within 15 units by means of the solution pool absolute gap and also within 1% of the incumbent by means of the solution pool relative gap. A solution is accepted in the pool only if it is valid for both the relative and the absolute gaps.

The solution pool absolute gap parameter can also be used as a stopping criterion for the populate procedure: if populate cannot enumerate any more solutions that satisfy this objective quality, then it will stop. In the presence of both an absolute and a relative solution pool gap parameter, populate will stop when the smaller of the two is reached.

Values

Any nonnegative real number; **default**: 1.0e+75.

limit on number of solutions kept in solution pool

Purpose

Limit on the number of solutions kept in the solution pool

Syntax

C Name	CPX_PARAM_SOLNPOOLCAPACITY (int)
C++ Name	SolnPoolCapacity (int)
Java Name	SolnPoolCapacity (int)
.NET Name	SolnPoolCapacity (int)
OPL Name	solnpoolcapacity
Interactive Optimizer	mip pool capacity
Identifier	2103

Description

Limits the number of solutions kept in the solution pool. At most, `SolnPoolCapacity` solutions will be stored in the pool. Superfluous solutions are managed according to the strategy set by the *solution pool replacement strategy* parameter (`SolnPoolReplace`, `CPX_PARAM_SOLNPOOLREPLACE`).

The optimization (whether by MIP optimization or the populate procedure) will not stop if more than `SolnPoolCapacity` solutions are generated. Instead, stopping criteria can be specified by these parameters:

- ◆ *limit on number of solutions generated for solution pool* (`PopulateLim`, `CPX_PARAM_POPULATELIM`)
- ◆ *relative gap for solution pool* (`SolnPoolGap`, `CPX_PARAM_SOLNPOOLGAP`)
- ◆ *absolute gap for solution pool* (`SolnPoolAGap`, `CPX_PARAM_SOLNPOOLAGAP`)
- ◆ *MIP node limit* (`NodeLim`, `CPX_PARAM_NODELIM`)
- ◆ *optimizer time limit* (`TiLim`, `CPX_PARAM_TILIM`)

The default value for `SolnPoolCapacity` is 2100000000, but it may be set to any nonnegative integer value. If set to zero, it will turn off all features related to the solution pool.

If you are looking for a parameter to control the number of solutions generated by the populate procedure, consider the parameter *limit on number of solutions generated for solution pool*.

Values

Any nonnegative integer; 0 (zero) turns off all features of the solution pool; **default:** 210000000.

relative gap for solution pool

Purpose

Relative gap for the solution pool

Syntax

C Name	CPX_PARAM_SOLNPOOLGAP (double)
C++ Name	SolnPoolGap (double)
Java Name	SolnPoolGap (double)
.NET Name	SolnPoolGap (double)
OPL Name	solnpoolgap
Interactive Optimizer	mip pool relgap
Identifier	2105

Description

Sets a relative tolerance on the objective value for the solutions in the solution pool. Solutions that are worse (either greater in the case of a minimization, or less in the case of a maximization) than the incumbent solution by this measure are not kept in the solution pool. For example, if you set this parameter to 0.01, then solutions worse than the incumbent by 1% or more will be discarded.

Values of the *absolute gap for solution pool* (SolnPoolAGap or CPX_PARAM_SOLNPOOLAGAP) and the *relative gap for solution pool* (SolnPoolGap or CPX_PARAM_SOLNPOOLGAP) may differ: For example, you may specify that solutions must be within 15 units by means of the solution pool absolute gap and within 1% of the incumbent by means of the solution pool relative gap. A solution is accepted in the pool only if it is valid for both the relative and the absolute gaps.

The solution pool relative gap parameter can also be used as a stopping criterion for the populate procedure: if populate cannot enumerate any more solutions that satisfy this objective quality, then it will stop. In the presence of both an absolute and a relative solution pool gap parameter, populate will stop when the smaller of the two is reached.

Values

Any nonnegative real number; **default:** 1.0e+75.

solution pool intensity

Purpose

Solution pool intensity

Syntax

C Name	CPX_PARAM_SOLNPOOLINTENSITY (int)
C++ Name	SolnPoolIntensity (int)
Java Name	SolnPoolIntensity (int)
.NET Name	SolnPoolIntensity (int)
OPL Name	solnpoolintensity
Interactive Optimizer	mip pool intensity
Identifier	2107

Description

Controls the trade-off between the number of solutions generated for the solution pool and the amount of time or memory consumed. This parameter applies both to MIP optimization and to the populate procedure.

Values from 1 (one) to 4 invoke increasing effort to find larger numbers of solutions. Higher values are more expensive in terms of time and memory but are likely to yield more solutions.

Effect

For MIP optimization, increasing the value of the parameter corresponds to increasing the amount of effort spent setting up the branch and cut tree to prepare for a subsequent call to the populate procedure.

For populate, increasing the value of this parameter corresponds, in addition, to increasing the amount of effort spent exploring the tree to generate more solutions. If MIP optimization is called before populate, populate will reuse the information computed and stored during MIP optimization only if this parameter has not been increased between calls. Similarly, if populate is called several times successively, populate will re-use the information computed and stored during previous calls to populate only if the solution pool intensity has not increased between calls. Therefore, it is most efficient **not** to change the value of this parameter between calls to MIP optimization and populate, nor between successive calls of populate. Increase the value of this parameter only if too few solutions are generated.

Settings

Its default value, 0 (zero), lets CPLEX choose which intensity to apply. If MIP optimization is called first after the model is read, CPLEX sets the intensity to 1 (one) for this call to MIP optimization and to subsequent calls of populate. In contrast, if populate is called directly after the model is read, CPLEX sets the intensity to 2 for this call and subsequent calls of populate.

For value 1 (one), the performance of MIP optimization is not affected. There is no slowdown and no additional consumption of memory due to this setting. However, populate will quickly generate only a small number of solutions. Generating more than a few solutions with this setting will be slow. When you are looking for a larger number of solutions, use a higher value of this parameter.

For value 2, some information is stored in the branch and cut tree so that it is easier to generate a larger number of solutions. This storage has an impact on memory used but does not lead to a slowdown in the performance of MIP optimization. With this value, calling populate is likely to yield a number of solutions large enough for most purposes. This value is a good choice for most models.

For value 3, the algorithm is more aggressive in computing and storing information in order to generate a large number of solutions. Compared to values 1 (one) and 2, this value will generate a larger number of solutions, but it will slow MIP optimization and increase memory consumption. Use this value only if setting this parameter to 2 does not generate enough solutions.

For value 4, the algorithm generates all solutions to your model. Even for small models, the number of possible solutions is likely to be huge; thus enumerating all of them will take time and consume a large quantity of memory. In this case, remember to set the *limit on number of solutions generated for solution pool* (PopulateLim, CPX_PARAM_POPULATELIM) to a value appropriate for your model; otherwise, the populate procedure will stop prematurely because of this stopping criterion instead of enumerating all solutions. In addition, a few limitations apply to this exhaustive enumeration, as explained in *Enumerating all solutions* in the *ILOG CPLEX User's Manual*.

Values

Value	Meaning
0	Automatic: let CPLEX choose; default
1	Mild: generate few solutions quickly
2	Moderate: generate a larger number of solutions
3	Aggressive: generate many solutions and expect performance penalty
4	Very aggressive: enumerate all practical solutions

solution pool replacement strategy

Purpose

Solution pool replacement strategy

Syntax

C Name	CPX_PARAM_SOLNPOOLREPLACE (int)
C++ Name	SolnPoolReplace (int)
Java Name	SolnPoolReplace (int)
.NET Name	SolnPoolReplace (int)
OPL Name	solnpoolreplace
Interactive Optimizer	mip pool replace
Identifier	2104

Description

Designates the strategy for replacing a solution in the solution pool when the solution pool has reached its capacity.

The value 0 (CPX_SOLNPOOL_FIFO) replaces solutions according to a first-in, first-out policy. The value 1 (CPX_SOLNPOOL_OBJ) keeps the solutions with the best objective values. The value 2 (CPX_SOLNPOOL_DIV) replaces solutions in order to build a set of diverse solutions.

If the solutions you obtain are too similar to each other, try setting SolnPoolReplace to 2.

The replacement strategy applies only to the subset of solutions created in the current call of MIP optimization or populate. Solutions already in the pool are not affected by the replacement strategy. They will not be replaced, even if they satisfy the criterion of the replacement strategy.

Values

Value	Symbol	Meaning
0	CPX_SOLNPOOL_FIFO	Replace the first solution (oldest) by the most recent solution; first in, first out; default
1	CPX_SOLNPOOL_OBJ	Replace the solution which has the worst objective
2	CPX_SOLNPOOL_DIV	Replace solutions in order to build a set of diverse solutions

MIP strong branching candidate list limit

Purpose

MIP strong branching candidate list limit

Syntax

C Name	CPX_PARAM_STRONGCANDLIM (int)
C++ Name	StrongCandLim (int)
Java Name	StrongCandLim (int)
.NET Name	StrongCandLim (int)
OPL Name	strongcandlim
Interactive Optimizer	mip limits strongcand
Identifier	2045

Description

Controls the length of the candidate list when CPLEX uses strong branching as the way to select variables. For more detail about that parameter, see *MIP variable selection strategy*:

- ◆ VarSel in the C++, Java, or .NET API;
- ◆ CPX_PARAM_VARSEL in the C API;
- ◆ set mip strategy variableselect 3 in the Interactive Optimizer.

Values

Any positive number; **default**: 10.

MIP strong branching iterations limit

Purpose

MIP strong branching iterations limit

Syntax

C Name	CPX_PARAM_STRONGITLIM (int)
C++ Name	StrongItLim (int)
Java Name	StrongItLim (int)
.NET Name	StrongItLim (int)
OPL Name	strongitlim
Interactive Optimizer	mip limits strongit
Identifier	2046

Description

Controls the number of simplex iterations performed on each variable in the candidate list when CPLEX uses strong branching as the way to select variables. For more detail about that parameter, see *MIP variable selection strategy*:

- ◆ VarSel in the C++, Java, or .NET API;
- ◆ CPX_PARAM_VARSEL in the C API;
- ◆ set mip strategy variableselect 3 in the Interactive Optimizer.

The default setting 0 (zero) chooses the iteration limit automatically.

Values

Value	Meaning
0	Automatic: let CPLEX choose; default
Any positive integer	Limit of the simplex iterations performed on each candidate variable

limit on nodes explored when a subMIP is being solved

Purpose

Limit on nodes explored when a subMIP is being solved

Syntax

C Name	CPX_PARAM_SUBMIPNODELIM (int)
C++ Name	SubMIPNodeLim (int)
Java Name	SubMIPNodeLim (int)
.NET Name	SubMIPNodeLim (int)
OPL Name	submipnodelim
Interactive Optimizer	mip limits submipnodelim
Identifier	2062

Description

Restricts the number of nodes explored when CPLEX is solving a subMIP. CPLEX solves subMIPs when it builds a solution from a partial MIP start, when repairing an infeasible MIP start, when executing the relaxation induced neighborhood search (RINS) heuristic, when branching locally, or when polishing a solution.

Values

Any positive integer; **default**: 500.

symmetry breaking

Purpose

Symmetry breaking

Syntax

C Name	CPX_PARAM_SYMMETRY (int)
C++ Name	Symmetry (int)
Java Name	Symmetry (int)
.NET Name	Symmetry (int)
OPL Name	symmetry
Interactive Optimizer	preprocessing symmetry
Identifier	2059

Description

Decides whether symmetry breaking reductions will be automatically executed, during the preprocessing phase, in a MIP model. The default level, -1, allows CPLEX to choose the degree of symmetry breaking to apply. The value 0 (zero) turns off symmetry breaking. Levels 1 through 5 apply increasingly aggressive symmetry breaking.

Values

Value	Meaning
-------	---------

-1	Automatic: let CPLEX choose; default
0	Turn off symmetry breaking
1	Exert a moderate level of symmetry breaking
2	Exert an aggressive level of symmetry breaking
3	Exert a very aggressive level of symmetry breaking
4	Exert a highly aggressive level of symmetry breaking
5	Exert an extremely aggressive level of symmetry breaking

global default thread count

Purpose

Global default thread count

Syntax

C Name	CPX_PARAM_THREADS (int)
C++ Name	Threads (int)
Java Name	Threads (int)
.NET Name	Threads (int)
OPL Name	threads
Interactive Optimizer	threads
Identifier	1067

Description

Sets the default number of parallel threads that will be invoked by any CPLEX parallel optimizer. Settings of this thread parameter interact with settings of the *parallel mode switch* (CPX_PARAM_PARALLELMODE, ParallelMode) as summarized in:

- ◆ **Table 1:** *Interaction of Callbacks with Threads and Parallel Mode Parameters: No Callbacks or only Informational Callbacks in Application*
- ◆ **Table 2:** *Interaction of Callbacks with Threads and Parallel Mode Parameters: Only Query Callbacks in Application*
- ◆ **Table 3:** *Interaction of Callbacks with Threads and Parallel Mode Parameters: Control Callbacks in Application*

For single threads, the parallel algorithms behave deterministically, regardless of thread and parallel mode parameter settings; that is, the algorithm proceeds sequentially in a single thread.

In this context, *sequential* means that the algorithm proceeds step by step, consecutively, in a predictable and repeatable order within a single thread. *Deterministic* means that repeated solving of the same model with the same parameter settings on the same computing platform will follow exactly the same solution path, yielding the same level of performance and the same values in the solution. Sequential execution is deterministic. In multithreaded computing, a deterministic setting requires synchronization between threads. *Opportunistic* entails less synchronization between threads and thus may offer better performance at the sacrifice of

repeatable, invariant solution paths and values in repeated runs on multiple threads or multiple processors.

In the following tables, *maximum number of threads* means the **minimum** of these two values:

- ◆ the maximum number of threads licensed;
- ◆ number of CPUs, cores available.

Interaction of Callbacks with Threads and Parallel Mode Parameters: No Callbacks or only Informational Callbacks in Application

	Parallel Mode Auto	Parallel Mode Opportunistic (-1)	Parallel Mode Deterministic (1)
Threads 0 Auto	Uses maximum number of threads; deterministic	Uses maximum number of threads; opportunistic	Uses maximum number of threads; deterministic
Threads 1	Uses one thread; sequential	Uses one thread; sequential	Uses one thread; sequential
Threads N > 1	Uses N threads; opportunistic	Uses N threads; opportunistic	Uses N threads; deterministic

Interaction of Callbacks with Threads and Parallel Mode Parameters: Only Query Callbacks in Application

	Parallel Mode Auto	Parallel Mode Opportunistic (-1)	Parallel Mode Deterministic (1)
Threads 0 Auto	Uses one thread; deterministic	Uses maximum number of threads; opportunistic	Uses maximum number of threads; deterministic
Threads 1	Uses one thread; sequential	Uses one thread; sequential	Uses one thread; sequential
Threads N > 1	Uses N threads; opportunistic	Uses N threads; opportunistic	Uses N threads; deterministic

Interaction of Callbacks with Threads and Parallel Mode Parameters: Control Callbacks in Application

	Parallel Mode Auto	Parallel Mode Opportunistic (-1)	Parallel Mode Deterministic (1)
Threads 0 Auto	Uses one thread; sequential	Uses one thread; sequential	Uses one thread; sequential
Threads 1	Uses one thread; sequential	Uses one thread; sequential	Uses one thread; sequential
Threads N > 1	Uses N threads; opportunistic	Uses N threads; opportunistic	Uses N threads; deterministic

Values

Value	Meaning
--------------	----------------

0	Automatic: let CPLEX decide; default
1	Sequential; single threaded
N	Uses N threads; N is limited by license and available processors

See also

parallel mode switch

optimizer time limit

Purpose

Optimizer time limit

Syntax

C Name	CPX_PARAM_TILIM (double)
C++ Name	TiLim (double)
Java Name	TiLim (double)
.NET Name	TiLim (double)
OPL Name	tilim
Interactive Optimizer	timelimit
Identifier	1039

Description

Sets the maximum time, in seconds, for a call to an optimizer. This time limit applies also to the conflict refiner.

The time is measured in terms of either CPU time or elapsed time, according to the setting of the *clock type for computation time* parameter (CPX_PARAM_CLOCKTYPE, ClockType).

The time limit for an optimizer applies to the sum of all its steps, such as preprocessing, crossover, and internal calls to other optimizers.

In a sequence of calls to optimizers, the limit is not cumulative but applies to each call individually. For example, if you set a time limit of 10 seconds, and you call mipopt twice then there could be a total of (at most) 20 seconds of running time if each call consumes its maximum allotment.

Values

Any nonnegative number; **default:** 1e+75.

See also

clock type for computation time

tree memory limit

Purpose

Tree memory limit

Syntax

C Name	CPX_PARAM_TRELIM (double)
C++ Name	TreLim (double)
Java Name	TreLim (double)
.NET Name	TreLim (double)
OPL Name	trelim
Interactive Optimizer	mip limits treememory
Identifier	2027

Description

Sets an absolute upper limit on the size (in megabytes, uncompressed) of the branch & cut tree. If this limit is exceeded, CPLEX terminates optimization.

Values

Any nonnegative number; **default:** 1e+75.

tuning information display

Purpose

Tuning information display

Syntax

C Name	CPX_PARAM_TUNINGDISPLAY (int)
C++ Name	TuningDisplay (int)
Java Name	TuningDisplay (int)
.NET Name	TuningDisplay (int)
OPL Name	tuningdisplay
Interactive Optimizer	tune display
Identifier	1113

Description

Specifies the level of information reported by the tuning tool as it works.

Use level 0 (zero) to turn off reporting from the tuning tool.

Use level 1 (one), the **default**, to display a minimal amount of information.

Use level 2 to display the minimal amount plus the parameter settings that the tuning tool is trying.

Use level 3 to display an exhaustive report of minimal information, plus settings that are being tried, plus logs.

Values

Value	Meaning
0	Turn off display
1	Display standard, minimal reporting; default
2	Display standard report plus parameter settings being tried
3	Display exhaustive report and log

tuning measure

Purpose

Tuning measure

Syntax

C Name	CPX_PARAM_TUNINGMEASURE
C++ Name	TuningMeasure
Java Name	TuningMeasure
.NET Name	TuningMeasure
OPL Name	tuningmeasure
Interactive Optimizer	tune measure
Identifier	1110

Description

Controls the measure for evaluating progress when a suite of models is being tuned.

Possible values are:

- ◆ CPX_TUNE_AVERAGE uses the mean average of time to compare different parameter sets over a suite of models.
- ◆ CPX_TUNE_MINMAX uses a minmax approach to compare the time of different parameter sets over a suite of models.

Values

Value	Meaning
CPX_TUNE_AVERAGE	mean time; default
CPX_TUNE_MINMAX	minmax time

tuning repeater

Purpose

Tuning repeater

Syntax

C Name	CPX_PARAM_TUNINGREPEAT (int)
C++ Name	TuningRepeat (int)
Java Name	TuningRepeat (int)
.NET Name	TuningRepeat (int)
OPL Name	tuningrepeat
Interactive Optimizer	tune repeat
Identifier	1112

Description

Specifies the number of times tuning is to be repeated on reordered versions of a given problem. The problem is reordered automatically by CPLEX permuting its rows and columns. This repetition is helpful when only one problem is being tuned, as repeated reordering and re-tuning may lead to more robust tuning results.

This parameter applies to only one problem in a tuning session. That is, in the Interactive Optimizer, this parameter is effective only when you are tuning a single problem; in the Callable Library (C API), this parameter is effective only when you are tuning a single problem with the routine `CPXtuneparam`.

Values

Any nonnegative integer; **default:** 1 (one)

tuning time limit

Purpose

Tuning time limit

Syntax

C Name	CPX_PARAM_TUNINGTILIM (double)
C++ Name	TuningTiLim (double)
Java Name	TuningTiLim (double)
.NET Name	TuningTiLim (double)
OPL Name	tuningtilim
Interactive Optimizer	tune timelimit
Identifier	1113

Description

Sets a time limit per model and per test set (that is, suite of models) applicable in tuning.

For an overall time limit on tuning, use the global time limit parameter (*optimizer time limit* TiLim, CPX_PARAM_TILIM).

For an example of how to use these time limit parameters together, see *Example: time limits on tuning in the Interactive Optimizer* in the *ILOG CPLEX User's Manual*.

Values

Any nonnegative number; **default**: 10 000.

See also

optimizer time limit

MIP variable selection strategy

Purpose

MIP variable selection strategy

Syntax

C Name	CPX_PARAM_VARSEL (int)
C++ Name	VarSel (int)
Java Name	VarSel (int)
.NET Name	VarSel (int)
OPL Name	varsel
Interactive Optimizer	mip strategy variableselect
Identifier	2028

Description

Sets the rule for selecting the branching variable at the node which has been selected for branching.

The minimum infeasibility rule chooses the variable with the value closest to an integer but still fractional. The minimum infeasibility rule (-1) may lead more quickly to a first integer feasible solution, but is usually slower overall to reach the optimal integer solution.

The maximum infeasibility rule chooses the variable with the value furthestest from an integer. The maximum infeasibility rule (1 one) forces larger changes earlier in the tree.

Pseudo cost (2) variable selection is derived from pseudo-shadow prices.

Strong branching (3) causes variable selection based on partially solving a number of subproblems with tentative branches to see which branch is the most promising. This strategy can be effective on large, difficult MIP problems.

Pseudo reduced costs (4) are a computationally less-intensive form of pseudo costs.

The default value (0 zero) allows CPLEX to select the best rule based on the problem and its progress.

Values

Value	Symbol	Meaning
-1	CPX_VARSEL_MININFEAS	Branch on variable with minimum infeasibility
0	CPX_VARSEL_DEFAULT	Automatic: let CPLEX choose variable to branch on; default
1	CPX_VARSEL_MAXINFEAS	Branch on variable with maximum infeasibility
2	CPX_VARSEL_PSEUDO	Branch based on pseudo costs
3	CPX_VARSEL_STRONG	Strong branching
4	CPX_VARSEL_PSEUDOREDUCED	Branch based on pseudo reduced costs

directory for working files

Purpose

Directory for working files

Syntax

C Name	CPX_PARAM_WORKDIR (string)
C++ Name	WorkDir (string)
Java Name	WorkDir (string)
.NET Name	WorkDir (string)
OPL Name	workdir
Interactive Optimizer	workdir
Identifier	1064

Description

Specifies the name of an existing directory into which CPLEX may store temporary working files, such as for MIP node files or for out-of-core barrier files. The default is the current working directory.

Values

Any existing directory; **default:** '.'

memory available for working storage

Purpose

Memory available for working storage

Syntax

C Name	CPX_PARAM_WORKMEM (double)
C++ Name	WorkMem (double)
Java Name	WorkMem (double)
.NET Name	WorkMem (double)
OPL Name	workmem
Interactive Optimizer	workmem
Identifier	1065

Description

Specifies an upper limit on the amount of central memory, in megabytes, that CPLEX is permitted to use for working memory before swapping to disk files, compressing memory, or taking other actions.

Values

Any nonnegative number, in megabytes; **default:** 128.0

See also

directory for working files

write level for MST, SOL files

Purpose

Write level for MST, SOL files

Syntax

C Name `CPX_PARAM_WRITELEVEL (int)`

C++ Name `WriteLevel (int)`

Java Name `WriteLevel (int)`

.NET Name `WriteLevel (int)`

OPL Name

Interactive Optimizer `write level i`

Identifier

Description

Sets the level of detail for CPLEX to write a solution to a file in SOL format or a MIP start to a file in MST format. CPLEX writes information about a MIP start to a formatted file of type MST with the file extension `.mst`. CPLEX writes information about a solution to a formatted file of type SOL with the file extension `.sol`. CPLEX records the write level at which it created a file in that file, so that the file can be read back accurately later.

The default setting of this parameter is 0 (zero) AUTO; that is, let CPLEX decide the level of detail. CPLEX behaves differently, depending on whether the format is SOL or MST and on whether it is writing a solution or MIP start. For SOL files, AUTO resembles level 1 (one): CPLEX writes all variables and their respective values to the file. For MST files, AUTO resembles level 2: CPLEX writes discrete variables and their respective values to the file.

When the value of this parameter is 1 (one), CPLEX writes **all** variables, both discrete and continuous, with their values.

When the value of this parameter is 2, CPLEX writes values for **discrete** variables only.

When the value of this parameter is 3, CPLEX writes values of **nonzero** variables only.

When the value of this parameter is 4, CPLEX writes values of **nonzero discrete** variables only.

Treatment of nonzeros

With respect to levels 3 and 4, where **nonzero** values are significant, CPLEX considers a value nonzero if the absolute value is strictly less than $1e-16$. In the case of **SOL** files, CPLEX applies this test to **primal** and **dual variable** values, that is, both x and pi variable values. In the case of **MST** files, CPLEX applies this test only to x values.

Restrictions due to reduced file size

Levels 3 and 4 reduce the size of files, of course. However, this reduced file entails restrictions and may create surprising results when the file is re-used. Levels 3 and 4 are not equivalent to levels 1 and 2. Indeed, if a MIP start does not contain a value for a variable expected at level 3 or 4, then this variable will be fixed to 0 (zero) when that MIP start file is processed. Specifically, at level 3, if the MIP start does not specify a value for a variable of any type, or at level 4, if the MIP start does not specify a value for a discrete variable, such a variable will be fixed to 0 (zero). Consequently, the same MIP start written at level 1 or 2 may produce satisfactory solutions, but the reduced MIP start file, written at level 3 or 4, perhaps does not lead to solutions. This surprising situation arises typically in the case of model changes with the addition of new variables.

Values

Value	Symbol	Meaning
0	AUTO	Automatic: let CPLEX decide
1	CPX_WRITELEVEL_ALLVARS	CPLEX writes all variables and their values
2	CPX_WRITELEVEL_DISCRETEVARS	CPLEX writes only discrete variables and their values
3	CPX_WRITELEVEL_NONZEROVARS	CPLEX writes only nonzero variables and their values
4	CPX_WRITELEVEL_NONZERODISCRETEVARS	CPLEX writes only nonzero discrete variables and their values

MIP zero-half cuts switch

Purpose

MIP zero-half cuts switch

Syntax

C Name	CPX_PARAM_ZEROHALFCUTS (int)
C++ Name	ZeroHalfCuts (int)
Java Name	ZeroHalfCuts (int)
.NET Name	ZeroHalfCuts (int)
OPL Name	zerohalfcuts
Interactive Optimizer	mip cuts zerohalfcut
Identifier	2111

Description

Decides whether or not to generate zero-half cuts for the problem. The value 0 (zero), the default, specifies that the attempt to generate zero-half cuts should continue only if it seems to be helping.

If you find that too much time is spent generating zero-half cuts for your model, consider setting this parameter to -1 (minus one) to turn off zero-half cuts.

If the dual bound of your model does not make sufficient progress, consider setting this parameter to 2 to generate zero-half cuts more aggressively.

Values

Value	Meaning
-1	Do not generate zero-half cuts
0	Automatic: let CPLEX choose; default
1	Generate zero-half cuts moderately
2	Generate zero-half cuts aggressively

ILOG CPLEX File Formats Reference Manual

This manual documents the file formats supported by ILOG CPLEX. It begins with a brief description of the file formats in alphabetic order. This manual continues with longer explanations of the following topics and formats.

ILOG CPLEX File Formats Reference Manual

This manual documents the file formats supported by ILOG CPLEX. It begins with a brief description of the file formats in alphabetic order. This manual continues with longer explanations of the following topics and formats.

In this section

Brief descriptions of file formats

Defines the file formats ILOG CPLEX recognizes and supports.

Reading and entering file formats in the Interactive Optimizer

Describes the file formats appropriate for the Interactive Optimizer

Saving problems in the Interactive Optimizer

Explains which format to use to save a problem in the Interactive Optimizer.

LP file format: matrix models

Summarizes the rules for the LP file format.

MPS file format: industry standard

Describes the industry standard MPS file format and ILOG CPLEX extensions to it.

Special records in MPS files: ILOG CPLEX extensions

ILOG CPLEX extends the MPS standard in several ways.

NET file format: network flow models

Describes NET file format.

PRM file format: parameter settings

Describes PRM file format for nondefault parameter settings.

BAS file format: advanced basis

Describes BAS file format to support advanced basis.

MST file format: MIP starts

Describes the MST file format for MIP starts.

ORD file format: priorities and branching orders

Describes ORD file format to support priorities and branching orders.

SOL file format: solution files

Describes SOL file format to support solution files.

FLT file format: filter files for the solution pool

Describes FLT file format to support filters of the solution pool.

CSV file format: comma separated values

Describes CSV file format for comma separated values.

XML file format: serialized models and solutions

Describes XML file format to support serialized models and solutions.

Brief descriptions of file formats

BAS

files are text files governed by Mathematical Programming System (MPS) conventions (that is, they are not binary) for saving a problem basis. They are documented in *BAS file format: advanced basis*.

BZ2

is not a file format specific to ILOG CPLEX. Rather, this file extension indicates that a file (possibly in one of the formats that ILOG CPLEX reads) has been compressed by BZIP2. On most platforms, ILOG CPLEX can automatically uncompress such a file and then read data from the file in one of the formats briefly described here

CLP

is the format ILOG CPLEX uses to represent the conflicting constraints and bounds (a subset of an infeasible model) that were found by the conflict refiner.

CSV

files contain comma-separated values. Concert Technology offers facilities in ILOG CPLEX for reading and writing such files. See the *Concert Technology Reference Manual* for details, especially the classes `IloCsvReader`, `IloCsvLine`, and `IloCsvReader::Iterator`.

DPE

is the format ILOG CPLEX uses to write a problem in a binary SAV file after the objective function of the problem has been perturbed.

DUA

format, governed by MPS conventions, writes the dual formulation of a problem currently in memory so that the MPS file can later be read back in and the dual formulation can then be optimized explicitly. This file format is largely obsolete now since you can use the command `set presolve dual` in the Interactive Optimizer to tell ILOG CPLEX to solve the dual formulation of an LP automatically. (You no longer have to tell ILOG CPLEX to write the dual formulation to a DUA file and then tell ILOG CPLEX to read the file back in and solve it.)

EMB

is the format ILOG CPLEX uses to save an embedded network it extracts from a problem. EMB files are written in MPS format.

FLT

is the format ILOG CPLEX uses to save filters for solution pools.

GZ

is not a file format specific to ILOG CPLEX. Rather, this file extension indicates that a file (possibly in one of the formats that ILOG CPLEX reads) has been compressed by `gzip`, the GNU zip program. On most platforms, ILOG CPLEX can automatically uncompress a gzipped file and then read data from a file in one of the formats briefly described here.

LP (Linear Programming)

is a ILOG CPLEX-specific file formatted for entering problems in an algebraic, row-oriented form. In other words, LP format allows you to enter problems in terms of their constraints. When you enter problems interactively in the Interactive Optimizer, you are implicitly using LP format. ILOG CPLEX also reads files in LP format. The section *LP file format: matrix models* describes the conventions and use of this format.

MIN

format for representing minimum-cost network-flow problems was introduced by DIMACS in 1991. More information about DIMACS network file formats is available via anonymous ftp from: `ftp://dimacs.rutgers.edu/pub/netflow/general-info/specs.tex`

MPS

is an industry-standard, ASCII-text file format for mathematical programming problems. This file format is documented in *MPS file format: industry standard*. Besides the industry conventions, ILOG CPLEX also supports extensions to this format for ILOG CPLEX-specific cases, such as names of more than eight characters, blank space as delimiters between columns, etc. The extensions are documented in *Special records in MPS files: ILOG CPLEX extensions*.

MST

is an XML format available with the ILOG CPLEX MIP optimizer. It is a text format ILOG CPLEX uses to enter a starting solution for a MIP. *MST file format: MIP starts* documents this file format.

NET

is a ILOG CPLEX-specific ASCII format for network-flow problems. It supports named nodes and arcs. *NET file format: network flow models* offers a fuller description of this file format.

ORD

is a format available with the ILOG CPLEX MIP optimizer. It is used to enter and to save priority orders for branching. It may contain branching instructions for individual variables. *ORD file format: priorities and branching orders* documents this file format.

PPE

is the format ILOG CPLEX uses to write a problem in a binary SAV file after the righthand side has been perturbed.

PRE

is the format ILOG CPLEX uses to write a presolved, reduced problem formulation to a binary SAV file. Since a presolved problem has been reduced, it does not correspond to the original problem.

PRM

is the format ILOG CPLEX uses to read and write nondefault values of parameters in a file. *PRM file format: parameter settings* documents the format and conventions for reading and writing such files through the Callable Library.

REW

is a format to write a problem in MPS format with disguised row and column names. This format is simply an MPS file format with all variable (column) and constraint (row) names converted to generic names. Variables are relabeled x_1 through x_n , and rows are renamed c_1 through c_m . This format may be useful, for example, for problems that you consider highly proprietary.

RLP

is the LP format using generic names in the Interactive Optimizer.

SAV

is a ILOG CPLEX-specific binary format for reading and writing problems and their associated basis information. ILOG CPLEX includes the basis in a SAV file only if the problem currently in memory has been optimized and a basis exists. This format offers the advantage of being numerically accurate (to the same degree as your platform) in contrast to text file formats that may lose numerical accuracy. It also has the additional benefit of being efficient with respect to read and write time. However, since a SAV file is binary, you cannot read nor edit it with your favorite text editor.

SOL

files are XML formatted files that contain solution information; they may also provide an advanced start for an optimization.

XML

as a file format is available to C++ users of Concert Technology to serialize models and solutions (that is, instances of `IloModel` and `IloSolution`). *XML file format: serialized models and solutions* explains more about this serialization API.

Reading and entering file formats in the Interactive Optimizer

The Interactive Optimizer accepts problems that you read in from files by means of the `read` command or that you enter interactively by means of the `enter` command. When you enter a problem interactively, ILOG CPLEX uses the LP file format; you may save the problem in any supported file format that you choose.

The `read` command of the ILOG CPLEX Interactive Optimizer accepts problem files in LP, MPS, and SAV formats. It also accepts basis files in BAS format. Problems previously saved in DUA, EMB, or REW formats are actually in MPS format. Presolved problems saved with the `pre` option are in SAV format. Problems in which the objective function has been perturbed and the problem saved with the `dpe` option are in SAV format. Problems in which the righthand side has been perturbed and the problem saved with the `ppe` option are in SAV format. Normally, ILOG CPLEX automatically detects which of these file types it is reading; you may also designate the correct file type if ILOG CPLEX does not detect the type automatically.

When ILOG CPLEX reads LP or MPS files, it automatically allocates enough physical memory (if available) to read the problem.

ILOG CPLEX reallocates memory automatically as it is reading from LP and MPS files so it usually **not** necessary to set values for the parameters `ColReadLim`, `RowReadLim`, and `NzReadLim`, but you can set these parameters to the problem sizes so no reallocations need be done. When ILOG CPLEX reads a SAV file, it is not necessary for you to reset these parameters. SAV files contain sufficient information about the size of the problem for ILOG CPLEX to allocate adequate space. For more information on the read limit parameters, see *variable (column) read limit*, *constraint (row) read limit*, and *nonzero element read limit* in the ILOG CPLEX *Parameters Reference Manual*.

Saving problems in the Interactive Optimizer

In the Interactive Optimizer, you save information about the problem currently in memory as a file in the LP, MPS, or SAV formats by means of the `write` command and its options. For a complete list of file formats that ILOG CPLEX supports, see *Brief descriptions of file formats*. Here are the options in the Interactive Optimizer for frequently used formats:

- ◆ Use the `bas` option to save a problem basis in MPS format.
- ◆ Use the `clp` option to write a conflict subproblem.
- ◆ Use the `flt` option to write filters for the solution pool.
- ◆ Use the `mst` option to write MIP start files. This option also applies to solutions in the solution pool. In fact, an additional option specifies whether to write a single solution specified by its index, or to write all solutions from the solution pool as MIP starts.
- ◆ Use the `pre` option to write a SAV file for the reduced, presolved problem formulation.
- ◆ Use the `prm` option to write a file of nondefault parameter settings.
- ◆ Use the `sol` option to write solution files. This option also applies to solutions in the solution pool. In fact, an additional option specifies whether to write a single solution specified by its index, or to write all solutions from the solution pool.

The SAV file format, because it is binary, is the format that preserves the greatest degree of precision in data. It can be effective in reducing read and write time for repetitively solved problems. However, because it is a binary format, it cannot be readily viewed or edited in standard text editors.

As a naming convention, we recommend that you use the file format for reading the file as the file extension when you write or save the file (for instance, `example.bas`, `example.lp`, `example.mps`, `example.sav`). When you follow this convention, ILOG CPLEX automatically recognizes the file type and eliminates additional prompts for you to specify a file type.

LP file format: matrix models

ILOG CPLEX provides a facility for entering a problem in a natural, algebraic LP formulation from the keyboard. The problem can be modified and saved from within ILOG CPLEX. This procedure is one way to create a file in a format that ILOG CPLEX can read. An alternative technique is to create a similar file using a standard text editor and to read it into ILOG CPLEX.

The ILOG CPLEX LP format is provided as an input alternative to the MPS file format. An LP format file may be easier to generate than an MPS file if your problem already exists in an algebraic format or if you have an application which generates the problem file more readily in algebraic format (such as a C application). *Working with LP files* in the *ILOG CPLEX User's Manual* in the chapter about managing input and output explains the implications of using LP format rather than MPS format.

ILOG CPLEX accepts any problem saved in an ASCII file provided that it adheres to the following syntax rules.

Comments

Anything that follows a backslash (\) is a comment and is ignored until a return is encountered. Blank lines are also ignored. Blank lines and comment lines may be placed anywhere and as frequently as you want in the file.

White space and line length

In general, white space between characters is irrelevant as it is skipped when a file is read. However, white space is not allowed in the keywords used to introduce a new section, such as MAX , MIN , ST , or BOUNDS . Also the keywords must be separated by white space from the rest of the file and must be at the beginning of a line. The maximum length for any name is 255. The maximum length of any line of input is 560.

Skipping spaces may cause ILOG CPLEX to misinterpret (and accept) an invalid entry, such as the following:

```
x1 x2 = 0
```

If the user intended to enter that example as a nonlinear constraint, ILOG CPLEX would instead interpret it as a constraint specifying that one variable named x_1x_2 must be equal to zero.

To indicate a quadratic constraint in this section, use explicit notation for multiplication and exponentiation (not space).

Problem sense

The problem statement must begin with the word `MINIMIZE` or `MAXIMIZE`, `MINIMUM` or `MAXIMUM`, or the abbreviations `MIN` or `MAX`, in any combination of upper- and lower-case characters. The word introduces the objective function section.

Variables

Variables can be named anything provided that the name does not exceed 255 characters, all of which must be alphanumeric (a-z, A-Z, 0-9) or one of these symbols: `! " # $ % & () , . ; ? @ _ ' { } ~`. Longer names are truncated to 255 characters. A variable name can not begin with a number or a period.

The letter `E` or `e`, alone or followed by other valid symbols, or followed by another `E` or `e`, should be avoided as this notation is reserved for exponential entries. Thus, variables can not be named `e9`, `E-24`, `E8cats`, or other names that could be interpreted as an exponent. Even variable names such as `eels` or `example` can cause a read error, depending on their placement in an input line.

Objective function

The objective function definition must follow `MINIMIZE` or `MAXIMIZE`. It may be entered on multiple lines as long as no variable, constant, or sense indicator is split by a return. For example, this objective function $1x_1 + 2x_2 + 3x_3$ can be entered like this:

```
1x1 + 2x2
+ 3x3
```

but not like this:

```
1x1 + 2x
2 + 3x3      \ a bad idea
```

because the second style splits the variable name `x2` with a return.

The objective function may be named by typing a name and a colon before the objective function. The objective function name and the colon must appear on the same line. Objective function names must conform to the same guidelines as variable names. (See the rule about *Variables*). If no objective function name is specified, ILOG CPLEX assigns the name `obj`.

An objective function may be quadratic. For an example and details about formatting a quadratic objective function, see the rule about *Quadratic terms*.

Constraints

The constraints section is introduced by the keyword `subject to`. This expression can also appear as `such that`, `st`, `S.T.`, or `ST.` in any mix of upper- and lower-case characters. One of these expressions must precede the first constraint and be separated from it by at least one space.

Each constraint definition must begin on a new line. A constraint may be named by typing a name and a colon before the constraint. The constraint name and the colon must appear on the same line. Constraint names must adhere to the same guidelines as variable names. (See the rule about names of *Variables*.) If no constraint names are specified, ILOG CPLEX assigns the names `c1`, `c2`, `c3`, etc.

The constraints are entered in the same way as the objective function; however, a constraint must be followed by an indication of its sense and a righthand side coefficient. The righthand side coefficient must be typed on the same line as the sense indicator. Acceptable sense indicators are `<`, `<=`, `=<`, `>`, `>=`, `=>`, and `=`. These are interpreted as `<`, `<=`, `<=`, `>`, `>=`, `>=`, and `=`, respectively.

For example, here is a named constraint:

```
time: x1 + x2 <= 10
```

Quadratic constraints are allowed in this section. Quadratic terms are specified inside square brackets `[]` as detailed in the rule about *Quadratic terms*. The specification of a quadratic constraint differs from the specification of a quadratic objective in one important way: in a quadratic constraint, the terms are not divided by two; that is, they are not multiplied by $1/2$, as they must be in a quadratic objective.

Indicator constraints are also allowed in this section. The rule about *MIP indicator constraints* explains how to specify indicator constraints.

Bounds

The optional `bounds` section follows the mandatory constraint section. It is preceded by the word `bounds` or `bound` in any mix of lower- and upper-case characters.

Each bound definition must begin on a new line. The format for a bound is $l_n \ x_n \ u_n$ except in the following cases.

Upper and lower bounds may also be entered separately as

$$l_n \ x_n$$
$$x_n \ u_n$$

with the default lower bound of 0 (zero) and the default upper bound of + remaining in effect until the bound is explicitly changed.

Bounds that fix a variable can be entered as simple equalities. For example, $x5 = 5.6$ is equivalent to $5.6 \leq x5 \leq 5.6$.

The bounds + (positive infinity) and - (negative infinity) must be entered as words: `+infinity, -infinity, +inf, -inf`.

A variable with a negative infinity lower bound and positive infinity upper bound may be entered as `free`, in any mix of upper- and lower-case characters, with a space separating the variable name and the word `free`. For example, `x7 free` is equivalent to $-infinity \leq x7 \leq +infinity$.

End of file

The file must end with the word `end` in any combination of upper- and lower-case characters, alone on a line, when it is created with the `enter` command. This word is not required for files that are read in to ILOG CPLEX, but it is strongly recommended. Files that have been corrupted can frequently be detected by a missing last line.

MIP integer variables

This rule applies to the ILOG CPLEX MIP optimizer.

To specify any of the variables as general integer variables, add a `GENERAL` section; to specify any of the variables as binary integer variables, add a `BINARY` section. The `GENERAL` and `BINARY` sections follow the `BOUNDS` section, if one is present; otherwise, they follow the constraints section. Either of the `GENERAL` or `BINARY` sections can precede the other. The `GENERAL` section is preceded by the word `GENERAL`, `GENERALS`, or `GEN` in any mix of upper- and lower-case characters which must appear alone on a line. The following line or lines should list the names of all variables which are to be restricted to general integer values, separated by at least one space. The `BINARY` section is preceded by the word `BINARY`, `BINARIES`, or `BIN` in any mix of upper- and lower-case characters which must appear alone on a line. The following line or lines should list the names of all variables which are to be restricted to binary integer values, separated by at least one space. Binary variables are automatically given bounds of 0 (zero) and 1 (one), unless alternative bounds are specified in the `BOUNDS` section, in which case a warning message is issued.

Here is an example of a problem formulation in LP format where `x4` is a general integer:

```
Maximize
  obj: x1 + 2 x2 + 3 x3 + x4
Subject To
  c1: - x1 + x2 + x3 + 10 x4 <= 20
  c2: x1 - 3 x2 + x3 <= 30
```

```
c3: x2 - 3.5 x4 = 0
Bounds
  0 <= x1 <= 40
  2 <= x4 <= 3
General
  x4
End
```

If branching priorities or branching directions exist, enter this information through ORD files, as documented in *ORD file format: priorities and branching orders*.

MIP semi-continuous variables

This rule applies to the ILOG CPLEX MIP optimizer.

To specify any of the variables as semi-continuous variables, that is as variables that may take the value 0 or values between the specified lower and upper bounds, use a SEMI-CONTINUOUS section. This section must follow the BOUNDS, GENERALS, and BINARIES sections. The SEMI-CONTINUOUS section is preceded by the keyword SEMI-CONTINUOUS, SEMI, or SEMIS. The following line or lines should list the names of all the variables which are to be declared semi-continuous, separated by at least one space.

```
Semi-continuous
x1 x2 x3
```

MIP special ordered sets

This rule applies to the ILOG CPLEX MIP optimizer. To specify special ordered sets, use an SOS section, which is preceded by the SOS keyword. The SOS section should follow the Bounds, General, Binaries and Semi-Continuous sections. Special ordered sets of type 1 require that, of the variables in the set, one at most may be nonzero. Special ordered sets of type 2 require that at most two variables in the set may be nonzero, and if there are two nonzeros, they must be adjacent. Adjacency is defined by the weights, which must be unique within a set given to the variables. The sorted weights define the order of the special ordered set. For MIP branch and cut, the order is used to decide how the variables are branched upon. See the *ILOG CPLEX User's Manual* for more information. The set is specified by an optional set name followed by a colon and then either of the S1 or S2 keywords (specifying the type) followed by a double colon. The set member names are listed on this line or lines, with their weights. Variable names and weights are separated by a colon, for example:

```
SOS
set1: S1:: x1:10 x2:13
```

MIP indicator constraints

This rule applies to ILOG CPLEX MIP optimizer.

To specify an indicator constraint, enter it among any other constraints in the model, like this:

```
[constraintname:] binaryvariable = value -> linear constraint
```

The constraint name, followed by a colon, is optional. The hyphen followed by the greater-than symbol (->), separates the indicator variable and its value from the linear constraint that is controlled. The indicator variable must be declared as a binary variable, and the value it is compared to must be either 0 (zero) or 1 (one).

Quadratic terms

This rule applies to applications licensed to solve problems with quadratic terms in them, that is, quadratic programming problems and quadratically constrained programs (QPs and QCPs). Quadratic coefficients may appear in the objective function. Quadratic coefficients may also appear in constraints under certain conditions. If there are quadratically constrained variables in the problem, see also rules about *Variables*, *Constraints*, and *Solving problems with quadratic constraints (QCP)* in the *ILOG CPLEX User's Manual*.

The algebraic coefficients of the function $x^T Q x$ are specified inside square brackets `[]`. The square brackets must be followed by a divide sign followed by the number 2. This convention denotes that all coefficients inside the square brackets will be divided by 2 in evaluating the quadratic terms of the objective function. All quadratic coefficients must appear inside square brackets. Multiple square bracket sections may be specified.

Inside of the square brackets, two variables are multiplied by an asterisk (*). For example, `[4x*y]` indicates that the coefficients of both of the off-diagonal terms of Q , corresponding to the variables x and y in the model are 2, since $4x*y$ equals $2x*y + 2x*y$. Each pair of off-diagonal terms of Q is specified only once. ILOG CPLEX automatically creates both off-diagonal entries of Q . Diagonal terms in Q (that is, terms with an exponent of 2) are indicated by the caret (^) followed by 2. For example, `4x^2` indicates that the coefficient of the diagonal term of Q corresponding to the variable x in the model is 4.

For example, this problem

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

subject to $a + b \leq 10$ and $a, b \geq 0$

in LP format looks like this:

```
Minimize
obj: a + b + [ a^2 + 4 a * b + 7 b^2 ]/2
```

```
Subject To  
c1: a + b >= 10  
End
```

Pools of lazy constraints and user-defined cuts

This rule is of interest only to advanced users.

It is possible to include pools of lazy constraints and user defined cuts in an LP file. A pool of lazy constraints or of user defined cuts must not contain any quadratic constraints. For more about these concepts, see *User-cut and lazy-constraint pools* in the *ILOG CPLEX User's Manual*.

MPS file format: industry standard

Describes the industry standard MPS file format and ILOG CPLEX extensions to it.

In this section

Overview of MPS

Introduces MPS file format.

Records in MPS format

Describes indicator records and data records in MPS file format.

Example of MPS file format

Shows formatted sample MPS file.

Overview of MPS

MPS format, long established on mainframe LP systems, has become a widely accepted standard for defining LP problems. In contrast to the ILOG CPLEX LP format, MPS format is a column-oriented format: problems are specified by column (variable) rather than by row (constraint).

Records in MPS format

MPS data files are analogous to a deck of computer input cards: each line of the MPS file represents a single card record. Records in an MPS data file consist of two types: indicator records and data records. The records contain fields delimited by blank spaces.

Indicator records

Indicator records separate the individual sections of the MPS file. Each indicator record contains a single word that begins in the first column. There are seven kinds of indicator records, each corresponding to sections of the MPS file. They are listed in *Indicator records*.

Indicator records

Section name/indicator record	Purpose
NAME	specifies the problem name; unlike other indicator records, the name record contains data
ROWS	specifies name and sense for each constraint
COLUMNS	specifies the name assigned to each variable (column) and the nonzero constraint coefficients corresponding to that variable
RHS	specifies the names of righthand side vectors and values for each constraint (row)
RANGES	specifies constraints that are restricted to lie in the interval between two values; interval endpoints are also specified
BOUNDS	specifies the limits within which each variable (column) must remain
ENDATA	signals the end of the data; always the last entry in an MPS file

Each section of the MPS file except the RANGES and BOUNDS sections is mandatory. If no BOUNDS section is present, all variables have their bounds set from 0 (zero) to + (positive infinity). Failure to include an RHS section causes ILOG CPLEX to generate a warning message and set all righthand side values to 0 (zero). Variables and constraints must be declared in the ROWS and COLUMNS sections before they are referenced in the RHS, RANGES, and BOUNDS sections.

Data records

Data records contain the information that describes the LP problem. Each data record comprises six fields, as in *Fields of a data record in MPS file format*. The fields must be separated by white space (that is, blank space, tab, etc.), and the first field must begin in column 2 or beyond. Not all fields are used within each section of the input file.

Fields of a data record in MPS file format

	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
Contents	Indicator	Name	Name	Value	Name	Value

Any ASCII character (32 through 126) is legal, but names must contain no embedded blanks. In addition, names over 255 characters are truncated. CPLEX issues an error message if truncation causes the names to lose their uniqueness. Numeric fields can be at most 25 characters long.

If the first character in Field 3 or 5 is a dollar sign (\$), the remaining characters in the record are treated as a comment. Another method for inserting comments is to place an asterisk (*) in column 1. Everything on such a line is treated as a comment.

Values may be defined with decimal or exponential notation and may utilize 25 characters. In exponential notation, plus (+) and minus (-) signs must precede the exponent value. If an exponent value is missing where one is expected, it is assigned a value of 0 (zero).

The ROWS section

In the ROWS section, each row of the problem is specified with its name and sense, one row per record.

Field 1 contains a single letter designating the sense of each row. Acceptable values are:

- ◆ N indicates a free row.
- ◆ G indicates a greater-than-or-equal-to row.
- ◆ L indicates a less-than-or-equal-to row.
- ◆ E indicates an equality row.

Field 2 contains a character identifier, maximum length of 255 characters, specifying the name of the row.

Fields 3-6 are not used in the ROWS section.

If more than one free row is specified, the first one is used as the objective function and the others are discarded.

The ROWS section of our example looks like this:

ROWS

```
N  obj
L  c1
L  c2
```

The COLUMNS section

In the COLUMNS section, all the columns of the constraint matrix are specified with their name and all of the nonzero elements. Multiple records may be required to completely specify a given column.

Field 1: Blank

Field 2: Column identifier

Field 3: Row identifier

Field 4: Value of matrix coefficient specified by Fields 2 and 3

Field 5: Row identifier (optional)

Field 6: Value of matrix coefficient specified by Fields 2 and 5 (optional)

After a matrix element is specified for a column, all other nonzero elements in that same column should be specified.

The COLUMNS section of our example looks like this:

COLUMNS

```
x1      obj      -1  c1      -1
x1      c2        1
x2      obj      -2  c1        1
x2      c2       -3
x3      obj      -3  c1        1
x3      c2        1
```

The RHS section

In the RHS section, the nonzero righthand-side values of the constraints are specified.

Field 1: Blank

Field 2: RHS identifier

Field 3: Row identifier

Field 4: Value of RHS coefficient specified by Field 2 and 3

Field 5: Row identifier (optional)

Field 6: Value of RHS coefficient specified by Field 2 and 5 (optional)

Several RHS vectors can exist. The name of each RHS vector appears in Field 2. However, only the first RHS vector is selected when a problem is read. Additional RHS vectors are discarded.

The RHS section of our example looks like this:

```
RHS
   rhs          c1                20   c2                30
```

The RANGES section

In the RANGES section, RHS range values to be applied to constraints may be specified.

Field 1: Blank

Field 2: Righthand side range vector identifier

Field 3: Row identifier

Field 4: Value of the range applied to row specified by Field 3

Field 5: Row identifier (optional)

Field 6: Value of the range applied to row specified by Field 5 (optional)

The effect of specifying a righthand side range depends on the sense of the specified row and whether the range has a positive or negative coefficient. *How range values are interpreted in data records of MPS files* specifies how range values are interpreted. For a given row, `rhs` is the righthand side value and `range` is the corresponding range value.

How range values are interpreted in data records of MPS files

Row type	Range value sign	Resulting rhs upper limit	Resulting rhs lower limit
G	+ or -	$\text{rhs} + \text{range} $	rhs
L	+ or -	rhs	$\text{rhs} - \text{range} $
E	+	$\text{rhs} + \text{range}$	rhs
E	-	rhs	$\text{rhs} + \text{range}$

The name of each range vector appears in Field 2. More than one range vector can be specified within an MPS file. However, only the first range vector is selected when a problem is read. Additional range vectors are discarded.

In our example, there are no ranged rows, but suppose we want to add the following constraint to our problem:

$$x_1 - 3x_2 + x_3 \geq 15$$

Instead of explicitly adding another row to the problem, we can represent this additional constraint by modifying row 2 of the example to make it a ranged row in this way:

$$15 \leq x_1 - 3x_2 + x_3 \leq 30$$

The RANGES section of the MPS file to support this modification looks like this:

```
RANGES
  rhs          c2          15
```

The name of each range vector appears in Field 2. However, only the first range vector is selected when a problem is read. Additional range vectors are discarded.

The BOUNDS section

In the BOUNDS section, bound values for variables may be specified.

Field 1: Type of bound. Acceptable values are:

- ◆ LO Lower bound
- ◆ UP Upper bound
- ◆ FX Fixed value (upper and lower bound the same)
- ◆ FR Free variable (lower bound - and upper bound +)
- ◆ MI Minus infinity (lower bound = -)
- ◆ PL Plus infinity (upper bound = +)

Field 2: Bound identifier

Field 3: Column identifier to be bounded

Field 4: Value of the specified bound

Fields 5 and 6 are not used in the BOUNDS section.

In our example, the BOUNDS section looks like this:

```
BOUNDS
UP BOUND    x1          40
```

If no bounds are specified, ILOG CPLEX assumes a lower bound of 0 (zero) and an upper bound of +. If only a single bound is specified, the unspecified bound remains at 0 or +, whichever applies, with one exception. If an upper bound of less than 0 is specified and no other bound is specified, the lower bound is automatically set to -. ILOG CPLEX deviates

slightly from a convention used by some MPS readers when it encounters an upper bound of 0 (zero). Rather than automatically set this variable's lower bound to - , ILOG CPLEX accepts both a lower and upper bound of 0, effectively fixing that variable at 0. ILOG CPLEX resets the lower bound to - only if the upper bound is less than 0. A warning message is issued when this exception is encountered.

More than one bound vector may exist. The name of each bound vector appears in Field 2. However, only the first bound vector is selected when a problem is read. Additional bound vectors are discarded.

Example of MPS file format

```
NAME          example2.mps
ROWS
  N  obj
  L  c1
  L  c2
COLUMNS
  x1      obj      -1  c1      -1
  x1      c2        1
  x2      obj      -2  c1        1
  x2      c2       -3
  x3      obj      -3  c1        1
  x3      c2        1
RHS
  rhs    c1        20  c2        30
BOUNDS
  UP BOUND  x1      40
ENDATA
```


Special records in MPS files: ILOG CPLEX extensions

ILOG CPLEX extends the MPS standard in several ways.

In this section

Overview of MPS extension

Introduces extensions to the MPS file format to support special features of ILOG CPLEX.

Objective sense and name in MPS files

Describes the ILOG CPLEX extension of MPS files to support name and sense of an objective function.

Integer variables in MPS files

Describes ILOG CPLEX extensions of MPS file format to support integer variables.

Special ordered sets (SOS) in MPS files

Describes ILOG CPLEX extensions of MPS file format to support special ordered sets.

Quadratic objective information in MPS files

Describes ILOG CPLEX extensions of MPS file format to support an objective function containing quadratic terms.

Quadratically constrained programs (QCP) in MPS files

Describes ILOG CPLEX extensions of MPS file format to support quadratically constrained programs.

Indicator constraints in MPS files

Describes ILOG CPLEX extensions of MPS file format to support indicator constraints.

User defined cuts in MPS files

Describes ILOG CPLEX extensions of MPS file format to support user defined cuts.

Lazy constraints in MPS files

Describes ILOG CPLEX extensions of MPS file format to support lazy constraints.

Overview of MPS extension

Historically, MPS format (including CPLEX MPS format for CPLEX version 2.1 and earlier releases) included restrictions, such as requiring input fields to occupy fixed columnar positions and limiting all names to a length of 8 characters or fewer. In CPLEX version 3.0 and subsequent releases, these restrictions were relaxed. The current ILOG CPLEX MPS format is actually an extended version of the historical MPS format. To allow for these extensions, certain practices which were accepted in MPS files for older CPLEX releases and other systems are no longer permitted. For example, since ILOG CPLEX no longer requires fixed columnar positions, blank spaces are interpreted as delimiters. Older MPS files containing names with embedded spaces therefore become unreadable. To maintain compatibility with earlier versions as well as MPS files from other systems, ILOG CPLEX provides an MPS file conversion utility which translates older files into the newer ILOG CPLEX MPS format. The section *Converting file formats* in the *ILOG CPLEX User's Manual* explains how to use the file conversion utility.

Objective sense and name in MPS files

ILOG CPLEX extends the MPS standard by allowing two additional sections: `OBJSEN` and `OBJNAME`. They may be specified after the `NAME` section. `OBJSEN` sets the objective function sense, and `OBJNAME` selects an objective function from among the free rows within the file. If neither of these sections appears in the MPS file, ILOG CPLEX assumes that the problem is a minimization and that the objective function is the first free row encountered in the `ROWS` section. If these options are used, they must appear in order and as the first and second sections after the `NAME` section. The values for `OBJSENSE` can be `MAX` or `MIN`.

Here is an example of these optional sections:

```
NAME          example.mps
OBJSENSE
  MAX
OBJNAME
  rowname
```

Integer variables in MPS files

If you use the ILOG CPLEX mixed integer optimizer, then you may restrict any or all variables to integer values. ILOG CPLEX accepts two commonly used ways of extending the MPS file format to include integer variables: in the COLUMNS section or in the BOUNDS section.

In the first way, integer variables are identified within the COLUMNS section of the MPS file by marker lines. A marker line is placed at the beginning and end of a range of integer variables. Multiple sets of marker lines are allowed. Integer marker lines have a field format consisting of Fields 2 through 4.

Field 2: Marker name

Field 3: 'MARKER' (including the single quotation marks)

Field 4: Keyword 'INTORG' and 'INTEND' to mark beginning and end respectively (including the single quotation marks)

Fields 5 and 6 are ignored.

The marker name must differ from the preceding and succeeding column names.

If no bounds are specified for the variables within markers, bounds of 0 (zero) and 1 (one) are assumed.

In the following example, column x4 is an integer variable and looks like this in the COLUMNS section of an MPS file, according to this first way of treating integer variables:

```
NAME
ROWS
 N  obj
 L  c1
 L  c2
 E  c3
COLUMNS
  x1      obj          -1   c1          -1
  x1      c2           1
  x2      obj          -2   c1           1
  x2      c2          -3   c3           1
  x3      obj          -3   c1           1
  x3      c2           1
  MARK0000 'MARKER'          'INTORG'
  x4      obj          -1   c1          10
  x4      c3          -3.5
  MARK0001 'MARKER'          'INTEND'
RHS
  rhs      c1          20   c2          30
BOUNDS
UP BOUND  x1          40
LO BOUND  x4           2
```

```
UP BOUND      x4          3
ENDATA
```

In the second way of treating integer variables, integer variables are declared in the BOUNDS section with special bound types in Field 1. The acceptable special bound types appear in *Special bound types for handling integer variables in MPS files*.

Special bound types for handling integer variables in MPS files

Type	Purpose	Special Considerations
BV	Binary variable	Field 4 must be 1.0 or blank
LI	Integer lower bound	Field 4 is the lower bound value and must be an integer
SC	Semi-continuous variable	Field 4 is the upper bound and must be specified
UI	Integer upper bound	Field 4 is the upper bound value and must be an integer

To specify general integers with no upper bounds, use LI with the value 0.0.

For example, column x4 is an integer variable declared in the BOUNDS section of an MPS file, according to this second way of treating integer variables:

```
NAME
ROWS
N  obj
L  c1
L  c2
E  c3
COLUMNS
  x1      obj      -1  c1      -1
  x1      c2        1
  x2      obj      -2  c1        1
  x2      c2       -3  c3        1
  x3      obj      -3  c1        1
  x3      c2        1
  x4      obj      -1  c1       10
  x4      c3      -3.5
RHS
  rhs      c1       20  c2       30
BOUNDS
UP BOUND  x1       40
LI BOUND  x4        2
UI BOUND  x4        3
ENDATA
```

Special ordered sets (SOS) in MPS files

If you use the ILOG CPLEX mixed integer optimizer (that is, the MIP optimizer), then you may define special ordered sets (SOS) in MPS format.

The convention for SOS uses set declaration lines and member declaration lines, both of which begin in column 2 or beyond. In a set declaration line, columns 2 and 3 specify S1 or S2. Optionally, the name of a set is specified in column 4. In a member declaration line, column 5 or beyond specifies a variable name. Note that in an MPS file, the SOS section must follow the BOUNDS section.

If weighting information is provided, it follows the member name in a member declaration line.

In the following example, an SOS section is placed after the BOUNDS section:

```
NAME
ROWS
  N  obj
  L  c1
  L  c2
  E  c3
COLUMNS
  x1      obj      -1  c1      -1
  x1      c2        1
  x2      obj      -2  c1        1
  x2      c2      -3  c3        1
  x3      obj      -3  c1        1
  x3      c2        1
  x4      obj      -1  c1       10
  x4      c3     -3.5
RHS
  rhs      c1        20  c2       30
BOUNDS
  UP BOUND  x1        40
  LI BOUND  x4         2
  UI BOUND  x4         3
SOS
  S1 set1
  x1          10000
  x2          20000
  x4          40000
ENDATA
```

'MARKER' lines for SOS in MPS files

'MARKER' lines are used to delimit SOS in the COLUMNS section of an MPS file, much like using integer markers. (The single quotation mark before and after the term is necessary.) The names of the sets are specified in the second field, starting in column 4 or beyond. Names of

sets must be unique. The 'MARKER' lines must come in pairs of an 'SOSORG' and 'SOSEND' surrounding the columns that are in the SOS. Optionally, in Field 1 of a 'MARKER' . . . 'SOSORG' line, either S1 or S2 may be specified to indicate the type of the SOS. An SOS 'MARKER' line without an S1 or S2 indicator is assumed to denote an S1 set. Members of an SOS may or may not be integer or binary variables.

There is no requirement that there be a constraint that all members of an SOS sum to 1.0 (nor is any such constraint implicit). However, providing such a constraint in your formulation may be desirable as it may strengthen the LP relaxation of the mixed integer problem, as for example in the case of an S1 set consisting of binary variables.

In the following example, the excerpt from the COLUMNS section of an MPS file defines an SOS Type 1 set consisting of x5 and x6, which may be continuous or integer variables.

```

S1 NAME1      'MARKER'                'SOSORG'
   x5         obj                    -9    c1      5
   x5         c2                      3
   x6         obj                    -6    c1      8
   x6         c3                    -4.5
NAME1      'MARKER'                'SOSEND'
```

The SOS 'MARKER' lines can appear between integer 'MARKER' lines (if all members of the SOS are integer), or integer 'MARKER' lines can appear between SOS 'MARKER' lines (if some members of the SOS are non-integer).

The MARKER format cannot accommodate overlapping SOSs. That is, a variable cannot be a member of two special ordered sets. Overlapping SOSs can, however, be specified by the ILOG CPLEX SOS format, documented in *Special ordered sets (SOS) in MPS files*.

REFROW section for SOS in MPS files

A REFROW section may be included immediately before the ROWS section. It consists of exactly one record line with the name of the reference row starting in Field 2. The specified row must also be defined in the ROWS section. The nonzeros of the reference row are used as weights within an SOS. All weights within one SOS must be unique values. A REFROW section is optional; if no reference row is specified, the weighting values 1, . . . , n is given to the n members of an SOS in the order in which they are read. In other words, without specific reference row information, it is assumed that the user has ordered the SOS variables in ascending order with respect to some relevant criterion (for example, in importance, capacity, objective weighting, or cost).

Quadratic objective information in MPS files

If you use the ILOG CPLEX barrier optimizer for quadratic programming problems (QPs), then you can specify quadratic objective coefficients in MPS format in a `QMATRIX` section.

Following the `BOUNDS` section, a `QMATRIX` section may be specified. Each line of this section defines one nonzero coefficient of the matrix Q . Each line should contain two variable names (which must have been specified in the `COLUMNS` section) in Fields 2 and 3, followed by a nonzero coefficient value in Field 4. For each off-diagonal coefficient, two lines must appear: one for the lower triangular element, and one for the upper triangular element. ILOG CPLEX evaluates the quadratic part of the objective function as $0.5 \times Qx$, when the coefficients of Q are specified in an MPS file.

For example, consider the following problem:

Minimize

$$a + b + 1/2 (a^2 + 4ab + 7b^2)$$

subject to

$$a + b = 10$$

$$a, b \geq 0$$

In MPS format, you may enter the problem in the following way:

```
NAME          problem
ROWS
  N  obj
  G  c1
COLUMNS
  a      obj      1  c1
  b      obj      1  c1
RHS
  rhs    c1      10
QMATRIX
  a      a        1
  a      b        2
  b      a        2
  b      b        7
ENDATA
```

You can also enter the quadratic objective coefficients by using a `QUADOBJ` section. In this format, only the upper diagonal elements of the Q matrix are entered. For the same example, the input with a `QUADOBJ` section looks like this:

```

NAME          problem
ROWS
  N  obj
  G  c1
COLUMNS
  a      obj      1  c1      1
  b      obj      1  c1      1
RHS
  rhs    c1      10
QUADOBJ
  a      a      1
  a      b      2
  b      b      7
ENDATA

```

If you have a model with quadratic objective information in MPS format in a `QUADOBJ` section of the following form, you do not have to convert your file in order for ILOG CPLEX to make use of it.

```
varname1 varname2 value
```

ILOG CPLEX can read that file and interpret the `QUADOBJ` section correctly. However, the MPS file writers of ILOG CPLEX do not produce a `QUADOBJ` section themselves. Instead, they produce a `QMATRIX` section, as explained here.

Quadratically constrained programs (QCP) in MPS files

As explained in the *ILOG CPLEX User's Manual in Solving problems with quadratic constraints (QCP)*, ILOG CPLEX can solve problems with quadratic terms among the constraints if the Q matrix for the quadratic term is positive semi-definite and the quadratic function defines a convex region. ILOG CPLEX has extended the MPS format to accommodate QCP models.

The quadratic constraints of such a model are listed in the ROWS section, and their linear coefficients appear in the COLUMNS section, just the same as coefficients from the linear constraints.

The quadratic terms go in QCMATRIX sections, one QCMATRIX per quadratic constraint. QCMATRIX sections appear after the optional SOS section. They may appear either after or before the QMATRIX (objective) section.

The name of the constraint appears on the same line after QCMATRIX .

The quadratic terms of the quadratic expression must be given as a symmetric matrix. That is, if there is an entry for Q_{ij} , then there must be an identical entry for Q_{ji} when i is not equal to j . This requirement is the same as for the QMATRIX section, where any quadratic terms in the objective function are declared. The formats of the Q parts are the same.

Indicator constraints in MPS files

Indicator constraints provide a way for you to express relations among variables by identifying a binary (0-1) variable to control whether or not a given constraint is active. ILOG CPLEX has extended the MPS format to express indicator constraints. The constraints to be controlled by the binary variable are listed in the ROWS section; their linear coefficients appear in the COLUMNS section (that is, the same as coefficients from linear constraints). Only rows of types E, L, and G may be part of indicator constraints. In other words, a row of type N cannot appear as a constraint controlled by a binary variable in this sense (that is, an indicator constraint).

The binary variables that control the linear constraints are specified in the BOUNDS section or with MARKER lines (that is, like any other binary variable). The relationship between the binary variables and the constraints they control is specified in the INDICATORS section. The INDICATORS section follows any quadratic constraint section and any quadratic objective section. Each line of the INDICATORS section has a type field starting in column 2 or beyond; the type must be "IF" followed by the name of the row of the indicator constraint, the name of the binary variable, and finally the value 0 (zero) or 1 (one) to indicate when the constraint should be active.

Rows that appear in the INDICATORS section cannot be ranged rows. In other words, a row that appears in the RANGES section cannot appear also in the INDICATORS section.

Here is an example of an INDICATORS section:

```
NAME          ind1.mps
ROWS
N  obj
L  row2
L  row4
E  row1
E  row3
COLUMNS
   x      obj          -1
   x      row2         1
   x      row4         1
   x      row1         1
   y      row4         1
   z      row4         1
   z      row3         1
RHS
   rhs    row2         10
   rhs    row4         15
BOUNDS
UI bnd    y           1
INDICATORS
IF row1   y           1
IF row3   y           0
```

```
ENDATA
```

That declaration represents the following model:

```
Minimize
  obj: - x
Subject To
  row2: x <= 10
  row4: x + y + z <= 15
  row1: y = 1 -> x = 0
  row3: y = 0 -> z = 0
Bounds
  0 <= y <= 1
Binaries
  y
End
```

User defined cuts in MPS files

The advanced feature user defined cuts can be declared in a special section following the ROWS section. The title of this section is USERCUTS . The order of sections must be ROWS USERCUTS . The format of the USERCUTS section is the same as the format of the ROWS section with this exception: the type must be one of E, L, or G; the row must not be ranged. For more information about user defined cuts, see *User-cut and lazy-constraint pools* in the *ILOG CPLEX User's Manual*.

Lazy constraints in MPS files

The advanced feature lazy constraints can be declared in a special section following the `ROWS` and `USERCUTS` sections. The title of this section is `LAZYCONS`. The order of sections must be `ROWS USERCUTS LAZYCONS`. The format of the `LAZYCONS` section is the same as the format of the `ROWS` section with this exception: the type must be one of `E`, `L`, or `G`; the row must not be ranged. For more information about lazy constraints and an example of an MPS file extended to include them, see *User-cut and lazy-constraint pools* in the *ILOG CPLEX User's Manual*.

NET file format: network flow models

The NET file format is an ASCII file format specific to ILOG CPLEX for network-flow problems. It is the recommended file format for representing pure network problems within CPLEX. This format is supported by Concert Technology, by the Callable Library, and by the Interactive Optimizer. In particular, it works with `CPXNETptr` objects (not `CPXLPptr` objects).

Comments

This is a free-format file; that is, line breaks or column positions are irrelevant to the interpretation of the file. The only exceptions to this convention are comments: anything from a backslash (\) character to the end of a line is a comment and does not contribute to the network specified by the file. Comments are allowed anywhere in the file.

Keywords

The NET format recognizes the following keywords in a file:

- ◆ MAXIMIZE
- ◆ MINIMIZE
- ◆ NETWORK
- ◆ ENDNETWORK
- ◆ SUPPLY
- ◆ DEMAND
- ◆ ARCS
- ◆ BOUNDS
- ◆ OBJECTIVE
- ◆ INFINITY
- ◆ FREE

Keywords are independent of character case. Keywords must be separated by white space from other symbols in the file.

White space

White space consists of one or more of the following:

- ◆ the space character
- ◆ the tab character (\t),
- ◆ the new line character (\n)
- ◆ a comment (that is, all characters following a backslash to the end of a line)

Abbreviations of keywords

Also, the NET format recognizes the abbreviations summarized in *Abbreviations of Keywords in NET File Format*.

Abbreviations of Keywords in NET File Format

Keyword	Abbreviation
INFINITY	INF
MINIMIZE	MIN
MAXIMIZE	MAX

Start of a NET file

A NET file must start with one of the following keywords:

- ◆ MAXIMIZE NETWORK
- ◆ MINIMIZE NETWORK

Both may be followed optionally by the name of a problem. If no name is specified, the filename will be used instead. This part of a NET file is referred to as the **start** of a NET file.

Names in a NET file

Names must follow the same conventions as they do for CPLEX LP format files. They must consist of a sequence of alphanumeric characters (a-z, A-Z, or 0-9) or one of the symbols: ! " # \$ % & () / , . : ; @ _ ' { } | ~. However, the first character may **not** be a digit or period (.). No names corresponding to the keywords are allowed. There is no restriction on the number of characters in a name within a NET file.

End of a NET file

The network specification of a NET file must end with the keyword `ENDNETWORK`. Anything following the keyword `ENDNETWORK` will be ignored. This keyword is referred to as the end of a NET file.

Sections of a NET file

Between its start and end, a NET file is divided into sections. Each section is introduced by its keyword and continues until the next section begins or the NET file ends. The keywords introducing sections are `SUPPLY`, `DEMAND`, `ARCS`, `BOUNDS`, and `OBJECTIVE`. Each section keyword may appear more than once in a NET file. They need not be in any order.

The SUPPLY section

In this section, supply values for nodes are specified. Each supply value is specified with the following sequence:

```
node-name : value
```

where `node-name` specifies the name of the node for which to set a supply value, and `value` is the value that will be assigned to node `node-name` as its supply value. If a node with this name does not already exist, a new node will be created with this name. If the node has been previously assigned a supply value, the new value overrides the previous value, and a warning will be issued.

The DEMAND section

This section corresponds to the `SUPPLY` section except that it specifies demand values instead of supply values. That is, instead of specifying a supply value `s` in the `SUPPLY` section, you can specify the negative of `s` in the `DEMAND` section and vice versa. The format for doing so is exactly the same: `node-name : value`. There is no requirement to use both a `SUPPLY` and a `DEMAND` section in a given model. You can fully specify any model using either of the section types alone by correctly using positive and negative values. The availability of either or both section types simply offers flexibility in model formulation.

The ARCS section

In this section, the arcs from-node (or tail) and to-node (or head) are specified. For each arc, the format is:

```
arc-name : from-node -> to-node
```

where `arc-name` specifies the name for the arc from `from-node` to `to-node`. If `arc-name` already exists, a warning message is issued, and the specified nodes override the previous ones. The nodes are referred to by node names. If a node does not yet exist, a new

node with this name will be created with supply value 0 (zero). Otherwise, the existing node of the specified name will be used.

The OBJECTIVE section

This section is used to assign objective values to arcs in the format:

```
arc-name : value
```

where `arc-name` must be the name of an arc that has previously been specified in an ARCS section. This arc will be assigned the objective value indicated by `value`. If an arc is assigned an objective value more than once, a warning message will be issued, and the most recently assigned objective value for that arc in the file will be used. If no objective value is specified for an arc, 0 (zero) will be used by default.

The BOUNDS section

In this section, bounds on the flow through an arc are specified in a variety of ways, similar to specifying bounds on variables in LP format. The general format is:

```
value1 <= arc-name <= value2
```

That general statement assigns a lower bound of `value1` and an upper bound of `value2` to the arc named `arc-name`. This arc must have previously been defined in an ARCS section.

Only one bound at a time may be specified for an arc. That is, the following are valid inputs: `value <= arc-name` to set the lower bound of the specified arc to `value` or `arc-name <= value` to set the upper bound of the specified arc to `value`. If the upper and lower bound for an arc are identical, you can write `arc-name = value` instead.

Bound values may be `INFINITY` or `-INFINITY`. An arc with lower bound `-INFINITY` and upper bound `INFINITY` may be entered as `FREE`, like this: `arc-name free`

If a bound is not specified for an arc, 0 (zero) will be used as the default lower bound and infinity as the default upper bound.

Example of NET file format

```
\ Except for this comment, this is the example network file
\ created by netex1.c
\
MINIMIZE NETWORK netex1
SUPPLY
    n1 : 20
    n4 : -15
    n5 : 5
    n8 : -10
ARCS
    a1 :      n1 ->      n2
    a2 :      n2 ->      n3
```

```
a3 :      n3 ->    n4
a4 :      n4 ->    n7
a5 :      n7 ->    n6
a6 :      n6 ->    n8
a7 :      n5 ->    n8
a8 :      n5 ->    n2
a9 :      n3 ->    n2
a10 :     n4 ->    n5
a11 :     n4 ->    n6
a12 :     n6 ->    n4
a13 :     n6 ->    n5
a14 :     n2 ->    n6
```

OBJECTIVE

```
a1 : 3
a2 : 3
a3 : 4
a4 : 3
a5 : 5
a6 : 6
a7 : 7
a8 : 4
a9 : 2
a10 : 6
a11 : 5
a12 : 4
a13 : 3
a14 : 6
```

BOUNDS

```
18 <= a1 <= 24
0 <= a2 <= 25
a3 = 12
0 <= a4 <= 10
0 <= a5 <= 9
a6 free
0 <= a7 <= 20
0 <= a8 <= 10
0 <= a9 <= 5
0 <= a10 <= 15
0 <= a11 <= 10
0 <= a12 <= 11
0 <= a13 <= 6
```

ENDNETWORK

PRM file format: parameter settings

It is possible to read and write a file of parameter settings with the Callable Library. This kind of file is known as a PRM file. The file extension for a PRM file is `.prm`. The Callable Library routine `CPXreadcopyparam` reads parameter values from a file with the `.prm` extension. The routine `CPXwriteparam` writes a file of the current nondefault parameter settings to a file with the `.prm` extension. Here is the format of such a file:

```
CPLEX Parameter File Version number
parameter_name    parameter_value
```

ILOG CPLEX reads the entire file before changing any of the parameter settings. After successfully reading a parameter file, the Callable Library first sets all parameters to their default value. Then it applies the settings it read from the parameter file. No changes are made if the parameter file contains errors, such as missing or illegal values. There is no checking for duplicate entries in the file. In the case of duplicate entries, the last setting in the file is applied.

When you write a parameter file from the Callable Library, only the nondefault values are written to the file. String values may be double-quoted or not, but are always written with double quotation marks.

Tip: The first line of a PRM file is significant to ILOG CPLEX. An easy way to produce a correctly formatted PRM file is to have ILOG CPLEX write it for you.

The comment character in a parameter file is `#`. ILOG CPLEX ignores the rest of the line.

The Callable Library issues a warning if the version recorded in the parameter file does not match the version of the product. A warning is also issued if a non-integral value is given for an integer-valued parameter.

Here is an example of such a file:

```
CPLEX Parameter File Version 11.0.0
CPX_PARAM_EPPER           3.450000000000000e-06
CPX_PARAM_OBJULIM        1.23456789012345e+05
CPX_PARAM_PERIND          1
CPX_PARAM_SCRIND          1
CPX_PARAM_WORKDIR        "tmp"
```

BAS file format: advanced basis

An MPS basis file, known as a BAS file, contains the information needed by ILOG CPLEX to define an advanced basis. Like an MPS file, the BAS file begins with a NAME indicator record and ends with an ENDATA record.

Tip: A BAS file is a text-based format that relies on each variable and each constraint having a name. If names do not exist, they will be created automatically, as needed, during a write operation. If you anticipate reading and writing BAS files, it is a good idea to assign a name to each variable (column) and to each constraint (row) yourself when you create the model.

A basis defines a list of basic structural variables and row variables. A structural variable is one of the variables (columns) defined in the MPS problem file. A row variable is actually the slack, surplus, or artificial variable associated with a row.

For linear programs, the total number of basic variables—both structural and row—is equal to the number of rows in the constraint matrix. Additionally, the number of basic structural variables is equal to the number of nonbasic row variables. By convention, an MPS basis file is built on the assumption that all row variables are basic and that all structural variables are nonbasic with values at their lower bound. The data records in a BAS file list structural and row variables that violate this assumption. This convention minimizes the size of the BAS file.

For quadratic programs, the total number of basic variables can exceed the number of rows and so not all basic variables can be paired with a nonbasic row variable.

Status indicators for variables in a BAS file

Value	Status
XU	Variable 1 is basic; variable 2 is nonbasic at its upper bound
XL	Variable 1 is basic; variable 2 is nonbasic at its lower bound
UL	Variable 1 is nonbasic and is at its upper bound
LL	Variable 1 is nonbasic and is at its lower bound
BS	Variable 1 is basic.

Field 1: Indicator specifying status of the named variables in Fields 2 and 3. Acceptable values appear in *Status indicators for variables in a BAS file*.

Field 2: Variable 1 identifier

Field 3: Variable 2 identifier (ignored if Field 1 is UL , LL or BS)

Variable 1 specifies a structural variable identifier which has entered the basis. By convention, this structural variable must displace one of the row variables. Variable 2 is a row variable that has left the basis. No relationship between structural variables entering the basis and row variables leaving the basis is implied within the BAS file.

In the *Example of MPS file format*, variables x_2 and x_3 are basic and the two constraints (row variables) are nonbasic. Also, x_1 was forced to its upper limit of 40. The optimal basis for that example appears in the following sample. ILOG CPLEX adds the number of iterations to the NAME record. The iteration count is useful if the basis file was automatically generated during a previously aborted run. The XL indicator in the first two data records indicates that x_3 and x_2 are basic and that the row variables for c_1 and c_2 are nonbasic at their lower bound. The third record shows that structural variable x_1 is nonbasic and at its upper bound.

```
NAME          example2.bas  Iterations 3  Rows 2  Cols 3
XL  x3        c1
XL  x2        c2
UL  x1
ENDATA
```

MST file format: MIP starts

If you use the ILOG CPLEX MIP optimizer, the MST file format is available to indicate MIP start values for specific variables, most commonly the integer variables. MST files are of the same format as SOL files. While SOL files have values for all variables, MST files written by ILOG CPLEX at the default write level of detail have values only for the integer variables and members of special ordered sets (SOS); at higher write levels of detail, MST files may also contain both discrete and continuous variables. For more information about the *write level for MST, SOL files*, see the *ILOG CPLEX Parameter Reference Manual* (`WriteLevel`, `CPX_PARAM_WRITELEVEL`).

MST file format also supports MIP starts from members of the solution pool. See details about the Concert Technology methods `cplex.writeMIPStarts` and the Callable Library routine `CPXmstwritemipstarts`, documented in their respective reference manuals.

ILOG CPLEX uses the start values in an MST file only if the advanced indicator parameter is on (that is, set to 1 (one) its default).

- ◆ In Concert Technology, use the method `IloCplex::setParam(AdvInd 1)`.
- ◆ In the Callable Library, use the routine `CPXsetintparam(env, CPX_PARAM_ADVIND, 1)`.
- ◆ In the Interactive Optimizer, use the command `set advance 1`.

Here is an example of an MST file:

```
<?xml version = "1.0" standalone="yes"?>
<?xml-stylesheet href="https://www.ilog.com/products/cplex/xmlv1.0/solution.
xsl" type="text/xsl"?>
<CPLEXSolution version="1.0">
  <header
    problemName="../../../../examples/data/mexample.mps"
    objectiveValue="-122.5"
    solutionTypeValue="3"
    solutionTypeString="primal"
    solutionStatusValue="101"
    solutionStatusString="integer optimal solution"
    MIPNodes="0"
    MIPIterations="3"/>
  <quality
    epInt="1e-05"
    epRHS="1e-06"
    maxIntInfeas="0"
    maxPrimalInfeas="0"
    maxX="40"
    maxSlack="2"/>
  <variables>
    <variable name="x4" index="3" value="3"/>
  </variables>
</CPLEXSolution>
```

```
</variables>  
</CPLEXSolution>
```

ORD file format: priorities and branching orders

If you use the ILOG CPLEX MIP optimizer, the ORD file format is available to indicate priority orders and branching directions for specific variables. Variables that are not given an explicit priority or that do not appear in an ORD file are assigned 0 (zero) priority. An ORD file begins with a NAME indicator record and ends with an ENDDATA record.

Integer variables are specified, one per line, with an optional branching direction (UP or DN) beginning in column 2 and 3. Names begin in column 5 or beyond. The variable name and its priority must be separated by one or more blank spaces.

Here is an example of an ORD file:

```
NAME
    x3                10
  DN x5                5
  UP x7
ENDDATA
```

ORD files created using CPLEX versions 2.1 or earlier used a fixed format in which the various data fields were limited to eight characters in length and restricted to specific columnar positions in each line. The extensions provided in the new ILOG CPLEX ORD file reader allow for more descriptive names and greater overall input flexibility. Most fixed-format ORD files conform to the new format. Any files that do not conform can be converted to the new format using the `convert` utility that comes with the standard ILOG CPLEX distribution. *Converting file formats* explains how to use that utility.

SOL file format: solution files

ILOG CPLEX enables you to read and write solution files, formatted in XML, for all problem types, for all application programming interfaces (APIs). These solution files, known as SOL files, carry the file extension `.sol`. The XML solution file format makes it possible for you to display and view these solution files in most browsers as well as to pass the solution to XML-aware applications. ILOG CPLEX also provides a stylesheet and schema in *yourCplexinstallation* /`include/ilcplex` to facilitate your use of this format in your applications.

- ◆ `solution.xsl` stylesheet

- ◆ `solution.xsd` schema

ILOG CPLEX can also read SOL files as an advanced start. SOL files contain basis statuses, if they are available, and solution values. The basis statuses can be used for advanced starts with simplex optimizers; the solution values can be used for a crossover from a barrier solution or as a MIP start from a mixed integer solution. A mixed integer solution may be from a conventional MIP optimization or from a member of the solution pool.

SOL files contain XML formatted information, as do MST files. Unlike MST files, at the default write level of detail, SOL files contain solution values for all variables rather than only the variables that define the integer feasible solution. SOL files are thus larger, take longer to input, and (in unusual cases of numerically difficult models) are less likely to provide a feasible starting point. Consequently, ILOG generally recommends using MST files to restart the optimization instead of SOL files. However, if you already use SOL files for other purposes, you can also use them to provide a feasible starting point.

SOL files also carry an optional name attribute, useful when the problem has names. SOL files also include an index, corresponding to the constraint index or variable index of the problem.

The SOL header gives information about the status of the solution. For example, the optimization status appears as a string and the numeric value of the ILOG CPLEX symbolic constant.

The SOL quality gives information about the quality of the solution. For example, the maximum primal infeasibility, the values of the tolerance parameters in effect during the optimization, and other quality information appears in this part.

There are, of course, methods and routines for reading and writing SOL files.

- ◆ In Concert Technology, use these methods:

- In the C++ API, see the methods `IloCplex::readSolution` and `IloCplex::writeSolution`.

- In the Java API, see the methods `IloCplex.readSolution` and `IloCplex.writeSolution`.
 - In the .NET API, see the methods `Cplex.ReadSolution` and `Cplex.WriteSolution`.
- ◆ In the Callable Library, use the routine `CPXreadcopysol` to read a SOL file and the routine `CPXsolwrite` to write SOL files.

Here is an example of a SOL file.

```
<?xml version = "1.0" standalone="yes"?>
<?xml-stylesheet href="https://www.ilog.com/products/cplex/xmlv1.0/solution.
xsl" type="text/xsl"?>
<CPLEXSolution version="1.1">
  <header
    problemName="../../../../examples/data/mexample.mps"
    solutionName="incumbent"
    solutionIndex="-1"
    objectiveValue="-122.5"
    solutionTypeValue="3"
    solutionTypeString="primal"
    solutionStatusValue="101"
    solutionStatusString="integer optimal solution"
    MIPNodes="0"
    MIPIterations="3"/>
  <quality
    epInt="1e-05"
    epRHS="1e-06"
    maxIntInfeas="0"
    maxPrimalInfeas="0"
    maxX="40"
    maxSlack="2"/>
  <linearConstraints>
    <constraint name="c1" index="0" slack="0"/>
    <constraint name="c2" index="1" slack="2"/>
    <constraint name="c3" index="2" slack="0"/>
  </linearConstraints>
  <variables>
    <variable name="x1" index="0" value="40"/>
    <variable name="x2" index="1" value="10.5"/>
    <variable name="x3" index="2" value="19.5"/>
    <variable name="x4" index="3" value="3"/>
  </variables>
</CPLEXSolution>
```

FLT file format: filter files for the solution pool

Describes FLT file format to support filters of the solution pool.

In this section

Overview of FLT

Introduces FLT file format for specifying filters of the solution pool.

Reading and writing filter files

Cites methods and routines for reading and writing filter files.

Syntax of a filter file

Describes the syntax of a filter file.

Diversity filters

Describes diversity filters for the solution pool.

Range filters

Describes range filters for the solution pool.

Overview of FLT

FLT denotes a file format for specifying filters (either diversity filters or range filters) associated with the solution pool of an application of ILOG CPLEX. This section documents that format.

For more general information about the solution pool, see *Solution pool: generating and keeping multiple solutions* of the *ILOG CPLEX User's Manual*. For an introduction to these filters and their purpose, along with examples of their use, see also *Diversity filters* and *Range filters* of the *ILOG CPLEX User's Manual*.

Reading and writing filter files

An existing filter file (such as one you create in your favorite text editor, or one you have saved from a previous session) can be added to your application and associated with the solution pool by one of these means:

- ◆ Concert Technology
 - `readFilters` in the C++ API
 - `readFilters(String)` in the Java API
 - `Cplex.ReadFilters` in the .NET API
- ◆ `CPXreadcopysolnpoolfilters` routine of the Callable Library;
- ◆ `read filename flt` command of the Interactive Optimizer.

The diversity and range filters already associated with a solution pool can be saved in a formatted file, as explained in *Filter files* in the *ILOG CPLEX User's Manual*.

- ◆ In Concert Technology,
 - in the C++ API, use the method `writeFilters`.
 - in the Java API, use the method `writeFilters(String)`.
 - in the .NET API, use the method `CPLEX.WriteFilters`.
- ◆ In the Callable Library, use the routine `CPXfltwrite`.
- ◆ `write filename flt` command of the Interactive Optimizer, where *filename* is the name of a file that the user supplies, and `flt` specifies the format of the file.

Syntax of a filter file

A filter file may contain one or more diversity filters, one or more range filters, or a combination of both types of filter.

The filters in a file are associated with a particular model. The name of the model follows the keyword `NAME`.

In a formatted filter file, the strings `-inf` and `inf` may be used to denote "no limit" in each of these contexts:

- ◆ the lower bound of a diversity or range filter,
- ◆ the upper bound of a diversity or range filter.

Here is a sample filter file. This filter is suitable for use with the model in *Example: simple facility location problem* in the *ILOG CPLEX User's Manual*. An explanation of this file appears in *Diversity filters* and *Range filters*.

```
NAME location
DIVFILTER f1 2 inf
x1 1.0 1
x2 1.0 1
x3 1.0 0
x4 1.0 0
RNGFILTER f2 -inf 0
transport 1.0
fixed -1.0
ENDATA
```

Diversity filters

A *diversity filter* allows you to control which solutions are generated and stored in the solution pool, according to their divergence from a reference solution. Only binary variables can be used to define a diversity filter.

The format of a diversity filter file lets you specify the names of the binary variables of interest, the weights to be assigned to those variables, and the reference values of those variables with which to compare all solutions. It also allows you to specify a lower bound and an upper bound on the divergence from the reference value(s).

The filter enforces the following constraint for a variable x :

```
lower bound <= diff(x) <= upper bound
```

This information is used to compute the *diversity measure*. The diversity measure is computed by summing the weighted absolute differences from the reference values, like this:

```
diff(x) = sum {weights[i] * |x[varind[i]] - refval[i]|};
```

The keyword `DIVFILTER` designates the beginning of a diversity filter in the file.

The name of the filter follows the keyword `DIVFILTER` on the same line. For example, in the sample in *Syntax of a filter file*, the name of the filter is `f1` and follows the keyword `DIVFILTER`.

The lower and upper bounds on the diversity function follow the name of the filter on the same line as the keyword `DIVFILTER`. In this example, the lower bound on diversity is 2, and the upper bound is infinity (that is, there is no limit).

Each successive line shows the name of a variable followed by the weight and the reference value for that variable.

In the example, equal weight (1.0) is given to the diversity of the four specified variables. The reference values are:

- ◆ 1 (one) for x_1 and x_2
- ◆ 0 (zero) for x_3 and x_4

Range filters

A *range filter* adds a constraint over a linear expression, like this:

```
lower bound <= linear expression <= upper bound
```

where the linear expression is a sum of weights multiplied by the value of a variable. That is,

```
sum{weights[i]*x[varind[i]]}
```

Range filters can be defined by any type of variables (binary, integer, continuous, semi-integer, semi-continuous).

In the sample filter file, the range filter corresponds to this constraint that the transportation cost must be no more than the fixed cost:

```
1.0 * transport - 1.0 * fixed <= 0
```

In a formatted FLT file, a range filter is specified by the keyword `RNGFILTER`.

The name of the filter (in this example, `f2`) follows the keyword `RNGFILTER` on the same line.

The lower and upper bounds on the linear expression follow the name of the filter on the same line as the keyword `RNGFILTER`. In this example, the lower bound on the expression is negative infinity (that is, no lower limit) and the upper bound is 0 (zero).

Each successive line shows the name of the variable and its coefficient in the linear expression.

CSV file format: comma separated values

ILOG CPLEX supports the file format known as CSV through XML facilities in Concert Technology. CSV is a file format consisting of lines of comma-separated values in ordinary ASCII text. Concert Technology 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* as the group `optim.concert.extensions`.

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.

XML file format: serialized models and solutions

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 *ILOG CPLEX 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.

Interactive Optimizer Commands

Lists the commands of the Interactive Optimizer.

Interactive Optimizer Commands

Lists the commands of the Interactive Optimizer.

In this section

Overview of commands

Introduces commands of the Interactive Optimizer.

Table of the commands of the Interactive Optimizer

Lists the commands of the Interactive Optimizer with links to samples or further documentation.

Managing parameters in the Interactive Optimizer

Describes access to parameters in the Interactive Optimizer.

Saving a parameter specification file

Describes purpose and use of a parameter specification file.

Overview of commands

This manual lists the commands of the Interactive Optimizer of ILOG CPLEX. For an introduction to the Interactive Optimizer, see the manual *Getting Started*, especially the tutorial for the Interactive Optimizer.

This manual begins with a table that lists Interactive Optimizer commands in alphabetic order with their primary options. For some commands, it also tells where examples of their use can be found in the *ILOG CPLEX User's Manual* or *Getting Started*.

These topics follow the table:

Table of the commands of the Interactive Optimizer

Interactive Optimizer Command		Options	Example
add			<i>Adding constraints and bounds in Getting Started</i>
baropt			<i>Using alternative optimizers in Getting Started</i>
baropt	dualopt		
baropt	primopt		
baropt	stop		
change	bounds		<i>Changing bounds in Getting Started</i>
change	coefficient		<i>Changing coefficients of variables in Getting Started</i>
change	delete		<i>Deleting entire constraints or variables in Getting Started</i>
change	delete	constraints	
change	delete	qconstraints	
change	delete	filters	
change	delete	indconstraints	
change	delete	mipstarts	<i>MIP starts and the Interactive Optimizer in User's Manual</i>
change	delete	solutions	
change	delete	sos	
change	delete	variables	
change	delete	equality	
change	delete	greater-than	
change	delete	less-than	

Interactive Optimizer Command		Options	Example
change	name		<i>Changing constraint or variable names in Getting Started</i>
change	objective		<i>Objective and RHS coefficients in Getting Started</i>
change	problem	<i>type</i>	<i>Using the MIP solution</i> <i>Changing problem type in QPs</i> <i>Diagnosing QP infeasibility in User's Manual</i>
change	problem	<i>fixed i</i>	<i>Accessing a solution in the solution pool in User's Manual</i>
change	qp term		<i>Changing quadratic terms in User's Manual</i>
change	rhs		<i>Objective and RHS coefficients in Getting Started</i>
change	sense		<i>Changing sense in Getting Started</i>
change	type		<i>Changing variable type in User's Manual</i>
change	values		<i>Changing small values to zero in Getting Started</i>
conflict			<i>Meet the conflict refiner in the Interactive Optimizer in User's Manual</i>
display	auxilliary	<i>filters</i>	displays names of filters associated with solution pool
display	auxilliary	<i>mipstarts</i>	displays names of MIP starts
display	auxilliary	<i>summary</i>	displays information about filters, MIP starts, priorities, bases
display	conflict	<i>all</i>	<i>Displaying a conflict in the Interactive Optimizer in User's Manual</i>
display	conflict	<i>constraints</i>	<i>Displaying a conflict in the Interactive Optimizer in User's Manual</i>
display	conflict	<i>indicators</i>	

Interactive Optimizer Command		Options	Example
display	conflict	qconstraints	
display	conflict	sos	
display	conflict	variables	<i>Displaying a conflict in the Interactive Optimizer in User's Manual</i>
display	problem	all	<i>Displaying a problem in Getting Started</i>
display	problem	binaries	<i>Interactive Optimizer display options for MIP problems in User's Manual</i>
display	problem	bounds	<i>Displaying bounds in Getting Started</i>
display	problem	constraints	<i>Displaying constraints in Getting Started</i>
display	problem	generals	<i>Interactive Optimizer display options for MIP problems in User's Manual</i>
display	problem	histogram	<i>Detecting and eliminating dense columns in User's Manual or Displaying a histogram of nonzero counts in Getting Started</i>
display	problem	indicators	
display	problem	integers	<i>Interactive Optimizer display options for MIP problems in User's Manual</i>
display	problem	names	<i>Displaying variable or constraint names in Getting Started</i>
display	problem	qconstraints	
display	problem	qpvariables	
display	problem	semi-continuous	
display	problem	sos	
display	problem	stats	<i>Solve the problem you intended or Interactive Optimizer display options for MIP problems in User's Manual</i>

Interactive Optimizer Command		Options	Example
display	problem	variable	
display	sensitivity	lb	<i>Performing sensitivity analysis in Getting Started</i>
display	sensitivity	objective	<i>Performing sensitivity analysis in Getting Started</i>
display	sensitivity	rhs	<i>Performing sensitivity analysis in Getting Started</i>
display	sensitivity	ub	<i>Performing sensitivity analysis in Getting Started</i>
display	settings		<i>Displaying parameter settings in Getting Started</i>
display	settings	all	<i>Displaying parameter settings in Getting Started</i>
display	settings	changed	<i>Displaying parameter settings in Getting Started</i>
display	solution	basis	
display	solution	bestbound	
display	solution	difference i j	<i>Examining the solution pool in User's Manual</i>
display	solution	dual	
display	solution	kappa	<i>Measuring problem sensitivity with basis condition number in User's Manual</i>
display	solution	list i n	<i>Examining the solution pool in User's Manual</i>
display	solution	member	<i>Examining the solution pool in User's Manual</i>
display	solution	objective	
display	solution	pool	<i>Examining the solution pool in User's Manual</i>

Interactive Optimizer Command		Options	Example
display	solution	qcslacks	
display	solution	quality	<i>Coping with an ill-conditioned problem or handling unscaled infeasibilities or Understanding solution quality from the barrier LP optimizer in User's Manual</i>
display	solution	reduced	
display	solution	slacks	<i>Displaying post-solution information in Getting Started</i>
display	solution	variables	<i>Displaying post-solution information in Getting Started</i>
display	solution number	i objective	<i>Examining the solution pool in User's Manual</i>
display	solution number	i qcslacks	
display	solution number	i quality	
display	solution number	i slacks	
display	solution number	i variables	
enter			<i>Entering a problem in Getting Started</i>
feasopt	constraints		<i>Invoking FeasOpt in User's Manual</i>
feasopt	variables		<i>Invoking FeasOpt in User's Manual</i>
feasopt	all		<i>Invoking FeasOpt in User's Manual</i>
help			<i>Using help in Getting Started</i>
mipopt			<i>Using the mixed integer optimizer in User's Manual</i>
netopt			<i>Example: network optimizer in the Interactive Optimizer or CPX_ALG_HYBNETOPT in Parameter settings for RootAlg and NodeAlg in User's Manual</i>

Interactive Optimizer Command		Options	Example
optimize			<i>Solving a problem in Getting Started</i>
populate			<i>Populating the solution pool in User's Manual</i>
primopt			<i>Using alternative optimizers in Getting Started</i>
quit			<i>Quitting ILOG CPLEX in Getting Started</i>
read	<i>filename</i>	<i>type</i>	<i>Starting from an advanced basis or Understanding the network log file or Filter files in User's Manual</i>
set	advance		<i>Starting from an advanced basis in User's Manual</i>
set	barrier		<i>Using the barrier optimizer in User's Manual</i>
set	barrier	algorithm	<i>Using the barrier optimizer or Choosing an ordering algorithm in User's Manual</i>
set	barrier	colnonzeros	<i>Detecting and eliminating dense columns in User's Manual</i>
set	barrier	convergetol	
set	barrier	crossover	
set	barrier	display <i>level</i>	<i>Using the barrier optimizer or Numeric instability due to elimination of too many dense columns in User's Manual</i>
set	barrier limits	corrections	<i>Change the limit on barrier corrections in User's Manual</i>
set	barrier limits	growth	
set	barrier limits	iterations	
set	barrier limits	objrange	<i>Difficulties with unbounded problems in User's Manual</i>

Interactive Optimizer Command		Options	Example
set	barrier limits	threads	
set	barrier	ordering	
set	barrier	qcpconvergetol	
set	barrier	startalg	
set	clocktype		
set	conflict	display <i>level</i>	
set	defaults		<i>Resetting defaults in Getting Started</i>
set	emphasis	memory	<i>Lack of memory or Memory emphasis: letting the optimizer use disk for storage in User's Manual</i>
set	emphasis	mip	<i>Emphasizing feasibility and optimality in User's Manual</i>
set	emphasis	numerical	<i>Numerical emphasis settings (LP) or Numerical emphasis settings (barrier) in User's Manual</i>
set	feasopt	tolerance	
set	logfile	<i>filename</i>	<i>Filling iteration logs in Getting Started</i>
set	lpmethod		
set	mip cuts	all	
set	mip cuts	<i>class</i>	<i>Parameters for controlling cuts in User's Manual</i>
set	mip cuts	cliques	
set	mip cuts	covers	
set	mip cuts	disjunctive	
set	mip cuts	flowcovers	
set	mip cuts	gomory	

Interactive Optimizer Command		Options	Example
set	mip cuts	gubcovers	
set	mip cuts	implied	
set	mip cuts	mircut	
set	mip cuts	pathcut	
set	mip cuts	zerohalf	<i>Zero-half cuts in User's Manual</i>
set	mip	display	
set	mip	interval	
set	mip limits	aggforcut	
set	mip limits	cutpasses	
set	mip limits	cutsfactor	<i>Parameters affecting cuts in User's Manual</i>
set	mip limits	gomorycand	
set	mip limits	gomorypass	
set	mip limits	nodes	<i>Parameters to limit MIP optimization in User's Manual</i>
set	mip limits	polishtime	
set	mip limits	populate	<i>Parameters of the solution pool in User's Manual</i>
set	mip limits	probetime	
set	mip limits	repairtries	
set	mip limits	solutions	<i>Parameters to limit MIP optimization in User's Manual</i>
set	mip limits	strongcand	
set	mip limits	strongit	
set	mip limits	strongthreads	

Interactive Optimizer Command		Options	Example
set	mip limits	submipnodelim	
set	mip limits	threads	
set	mip limits	treememory	<i>Reset the tree memory parameter in User's Manual</i>
set	mip	ordtype	
set	mip pool	absgap	<i>Parameters of the solution pool in User's Manual</i>
set	mip pool	capacity	<i>Parameters of the solution pool in User's Manual</i>
set	mip pool	intensity	<i>Parameters of the solution pool in User's Manual</i>
set	mip pool	relgap	<i>Parameters of the solution pool in User's Manual</i>
set	mip pool	replace	<i>Parameters of the solution pool in User's Manual</i>
set	mip strategy	backtrack	<i>Parameters for controlling branch & cut strategy in User's Manual</i>
set	mip strategy	bbinterval	<i>Parameters for controlling branch & cut strategy in User's Manual</i>
set	mip strategy	branch	<i>Parameters for controlling branch & cut strategy in User's Manual</i>
set	mip strategy	dive	
set	mip strategy	file	
set	mip strategy	heuristicfreq	<i>Heuristics in User's Manual</i>
set	mip strategy	lbheuristic	

Interactive Optimizer Command		Options	Example
set	mip strategy	nodeselect	<i>Parameters for controlling branch & cut strategy in User's Manual</i>
set	mip strategy	order	
set	mip strategy	presolvenode	
set	mip strategy	probe	<i>Probing in User's Manual</i>
set	mip strategy	rinsheur	<i>Relaxation induced neighborhood search (RINS) heuristic in User's Manual</i>
set	mip strategy	search	<i>MIP dynamic search switch in Parameters Reference Manual</i>
set	mip strategy	startalgorithm	
set	mip strategy	subalgorithm	<i>NodeAlg parameter and difficult subproblems in User's Manual</i>
set	mip strategy	variableselect	<i>Parameters for controlling branch & cut strategy in User's Manual</i>
set	mip tolerances	absmipgap	
set	mip tolerances	integrality	
set	mip tolerances	lowercutoff	
set	mip tolerances	mipgap	
set	mip tolerances	objdifference	<i>Time wasted on overly tight optimality criteria in User's Manual</i>
set	mip tolerances	relobjdifference	
set	mip tolerances	uppercutoff	
set	network	display	<i>Understanding the network log file in User's Manual</i>
set	network	iterations	<i>Limiting iterations in the network optimizer in User's Manual</i>

Interactive Optimizer Command		Options	Example
set	network	netfind	
set	network	pricing	<i>Selecting a pricing algorithm for the network optimizer in User's Manual</i>
set	network tolerances	feasibility	<i>Controlling tolerance in User's Manual</i>
set	network tolerances	optimality	
set	output	<i>channel</i>	
set	output	mpslong	
set	output	logonly	<i>Interpreting solution quality in User's Manual</i>
set	parallel	<i>mode</i>	<i>Using parallel optimizers in the Interactive Optimizer in User's Manual</i>
set	preprocessing	aggregator	<i>Preprocessing</i> <i>Preprocessing and memory requirements</i> <i>Parameters for controlling MIP preprocessing in User's Manual</i>
set	preprocessing	boundstrength	<i>Parameters for controlling MIP preprocessing in User's Manual</i>
set	preprocessing	coeffreduce	<i>Parameters for controlling MIP preprocessing</i> <i>Examples: optimizing a simple MIP problem in User's Manual</i>
set	preprocessing	dependency	<i>Preprocessing (continuous)</i> <i>Preprocessing (discrete) in User's Manual</i>
set	preprocessing	dual	<i>Using a starting-point heuristic in User's Manual</i>
set	preprocessing	fill	<i>Preprocessing in User's Manual</i>

Interactive Optimizer Command		Options	Example
set	preprocessing	linear	
set	preprocessing	numpass	
set	preprocessing	presolve	<i>Preprocessing and memory requirements (continuous) in User's Manual</i> <i>Parameters for controlling MIP preprocessing in User's Manual</i>
set	preprocessing	qpmakepsd	
set	preprocessing	reduce	<i>Preprocessing (continuous) or Preprocessing and feasibility (discrete) in User's Manual</i>
set	preprocessing	relax	<i>Parameters for controlling MIP preprocessing in User's Manual</i>
set	preprocessing	repeatpresolve	<i>Preprocessing: presolver and aggregator (discrete) in User's Manual</i>
set	preprocessing	symmetry	
set	qpmethod		
set	read	constraints	
set	read	datacheck	<i>Displaying problem statistics in Getting Started</i>
set	read	nonzeroes	
set	read	qpnzeroes	
set	read	scale	<i>Scaling in User's Manual</i>
set	read	variables	
set	sifting	algorithm	
set	sifting	display	
set	sifting	iterations	

Interactive Optimizer Command		Options	Example
set	simplex	crash	<i>Crash parameter settings for the primal simplex optimizer in User's Manual</i>
set	simplex	dgradient	
set	simplex	display	
set	simplex limits	iterations	
set	simplex limits	lowerobj	
set	simplex limits	perturbation	<i>Stalling due to degeneracy in User's Manual</i>
set	simplex limits	singularity	<i>Repeated singularities in User's Manual</i>
set	simplex limits	upperobj	
set	simplex	perturbation	<i>Stalling due to degeneracy in User's Manual</i>
set	simplex	pgradient	
set	simplex	pricing	
set	simplex	refactor	<i>Refactoring frequency and memory requirements in User's Manual</i>
set	simplex tolerances	feasibility	<i>Maximum bound infeasibility: identifying largest bound violation in User's Manual</i>
set	simplex tolerances	markowitz	<i>Inability to stay feasible in User's Manual</i>
set	simplex tolerances	optimality	<i>Maximum reduced-cost infeasibility in User's Manual</i>
set	threads		
set	timelimit		<i>Parameters to limit MIP optimization in User's Manual</i>

Interactive Optimizer Command		Options	Example
set	workdir	<i>prompt for directory</i>	<i>Memory emphasis: letting the optimizer use disk for storage in User's Manual</i>
set	workmem	<i>prompt for new value of working memory available</i>	<i>Memory emphasis: letting the optimizer use disk for storage or Parameters to limit MIP optimization in User's Manual</i>
tranopt			
tune	display	<i>i</i>	<i>Tuning tool in User's Manual</i>
tune	filenames	<i>parameterfile .prm</i>	<i>Example: time limits on tuning in the Interactive Optimizer and Fixing parameters and tuning multiple models in the Interactive Optimizer in User's Manual</i>
write	filenames	<i>type</i>	<i>Preprocessing (continuous) or Repeated singularities or Difficulty solving subproblems: overcoming degeneracy or MIP starts and the Interactive Optimizer or Saving QP problems in User's Manual</i>
xecute	command		<i>Executing operating system commands in Getting Started</i>

Managing parameters in the Interactive Optimizer

To see the current value of a parameter that interests you in the Interactive Optimizer, use the command `display settings`. The command `display settings changed` lists only those parameters where the value is not the default value. The command `display settings all` lists all parameters and their values.

To change the value of a parameter in the Interactive Optimizer, use the command `set` followed by options to indicate the parameter and the value you want it to assume.

In the reference manual of ILOG CPLEX Parameters, you will find the name of each parameter and its options in the Interactive Optimizer, along with the name of the parameter in Concert Technology and the Callable Library. That manual also describes the purpose of each parameter and documents its possible settings.

In the reference manual of the ILOG CPLEX Callable Library, the group `optim.cplex.manageparameters` documents the Callable Library routines that access parameters.

Saving a parameter specification file

You can tell the ILOG CPLEX Interactive Optimizer to read customized parameter settings from a parameter specification file. By default, ILOG CPLEX expects a parameter specification file to be named `plex.par`, and it looks for that file in the directory where it is executing. However, you can rename the file, or tell ILOG CPLEX to look for it in another directory by setting the system environment variable `CPLXPFILE` to the full path name of your parameter specification file. You set that environment variable in the customary way for your platform. For example, on a UNIX platform, you might use a shell command to set the environment variable, or on a personal computer running Microsoft Windows, you might click on the System icon in the control panel, then select the environment tab from the available system properties tabs, and then define the variable there.

During initialization in the Interactive Optimizer, ILOG CPLEX locates any available parameter specification file (by checking the current execution directory for `plex.par` and by checking the environment variable `CPLXPFILE`) and reads that file. As it opens the file, ILOG CPLEX displays the message “Initial parameter values are being read from `plex.par`” (or from the parameter specification file you specified). As ILOG CPLEX displays that message on the screen, it also writes the message to the log file. If ILOG CPLEX cannot open the file, it displays no message, records no note in the log file, and uses default parameter settings.

You can use a parameter specification file to change any parameter or parameters accessible by the `set` command in the Interactive Optimizer. The parameter types, names, and options are those used by the `set` command in the Interactive Optimizer.

To create a parameter specification file, you can use either of these alternatives:

- ◆ Use an ordinary text editor to create a file where each line observes the following syntax:

parameter-name option value

- ◆ Use the command `display settings` in the Interactive Optimizer to generate a list of current parameter settings. Those settings will be recorded in the log file. You can then edit the log file with your preferred text editor to create your parameter specification file.

`display settings changed` lists parameters different from the default with their values.

`display settings all` lists all parameters with their values.

Each entry on a line must be separated by at least one space or tab. Blank lines in a parameter specification file are acceptable; there are no provisions for comments in the file. You may abbreviate parameter names to unique character sequences, as you do in the `set` command.

As ILOG CPLEX reads a parameter specification file, if the parameter name and value are valid, ILOG CPLEX sets the parameter and writes a message about it to the screen and to the

log file. If ILOG CPLEX encounters a repeated parameter, it uses the last value specified. ILOG CPLEX terminates under the following conditions:

- ◆ if it encounters a parameter that is unknown;
- ◆ if it encounters a parameter that is not unique;
- ◆ if the parameter is correctly specified but the value is missing, invalid, or out of range.

Here is an example of a parameter specification file that tells ILOG CPLEX to use wall clock rather than CPU time while limiting total run time to 60 seconds. It also instructs ILOG CPLEX to open a log file named `problem.log`.

```
clocktype 2
timelimit 60
logfile   problem.log
```


*Index***Symbols**

- <CPLEXSolutionPool> removed **41**
- <CPLEXSolutions> new XML element **41**

A

- absolute gap
 - solution pool **1471**
- absolute objective difference **1424**
 - in integrality constraints of a MIP **824**
 - in MIP performance **890**
- absolute optimality tolerance
 - definition **890**
 - gap **890**
- accessing
 - basic rows and columns of solution in Interactive Optimizer **199**
 - basis information (C++ API) **267, 400**
 - current parameter value (C API) **541**
 - current parameter value (C++ API) **395**
 - default parameter value (C API) **541**
 - dual values (C++ API) **399**
 - dual values in Interactive Optimizer **199**
 - dual values in Interactive Optimizer (example) **199**
 - maximum parameter value (C API) **541**
 - maximum parameter value (Java API) **453**

- minimum parameter value (C API) **541**
- minimum parameter value (Java API) **453**
- objective function value (C++ API) **399**
- objective function value in Interactive Optimizer **199**
- quality of solution in Interactive Optimizer **199**
- reduced cost (Java API) **292**
- reduced costs (C++ API) **399**
- reduced costs in Interactive Optimizer **199**
- slack values in Interactive Optimizer **199**
- solution quality (C++ API) **403**
- solution values (C++ API) **251, 399**
- solution values in Interactive Optimizer **199**
- accessing incumbent (.NET API) **49**
- accessing incumbent (C API) **51**
- accessing incumbent (C++ API) **46**
- accessing incumbent (Java API) **47**
- active model
 - as instance of IloCplex (Java API) **438**
 - MIP (Java API) **447**
- active node **1153**
- add Interactive Optimizer command **218**
 - file name and **218**

- syntax **219**
- add method
 - IloModel C++ class
 - extensible arrays **386**
 - modifying a model **406**
- add(obj) method (Java API) **289**
- adding
 - bounds in Interactive Optimizer **218**
 - constraint to model (C++ API) **272**
 - constraints in Interactive Optimizer **218**
 - from a file in Interactive Optimizer **218**
 - interactively in Interactive Optimizer **218**
 - objective (shortcut) (Java API) **289**
 - objective function to model (C++ API) **247**
 - rows to a problem (C API) **345**
- addLe method (Java API) **296**
- addMinimize method (Java API) **289, 296, 438**
- advanced basis
 - advanced start indicator in Interactive Optimizer **197**
 - example **688**
 - ignored by tuning tool **639**
 - in networks **654**
 - parallel threads and **651**
 - primal feasibility and **651**

- reading from file (LP) **664**
- saving to file (LP) **664**
- starting from (LP) **664**
- advanced start **1329**
 - barrier and **1329**
 - basis and **1329**
 - example (LP) **688**
 - node exploration limit **1480**
 - presolve and **1329**
 - repair tries **1455**
 - root algorithm and **1458**
- Advanced start switch **90**
- AdvInd **90, 1329**
- AdvInd parameter
 - MIP start **868**
 - solution polishing and **857**
- AggCutLim **850, 1331**
- AggCutLim parameter
 - controlling cuts **850**
- AggFill **1332**
- aggregation limit **1331**
- aggregator
 - barrier preprocessing **718**
 - simplex and **661**
- algorithm
 - choosing in LP (C++ API) **392**
 - controlling in IloCplex (C++ API) **395**
 - creating object (C++ API) **250**
 - pricing **666**

- role in application (C++ API) **254**
- type (Java API) **449**
- using multiple **1071**
- Algorithm.Barrier (Java API) **461**
- and method (Java API) **297**
- application
 - and Callable Library **132**
 - and Concert Technology **132**
 - compiling and linking (C++ API) **241**
 - compiling and linking Callable Library (C API) **319**
 - compiling and linking Component Libraries **150**
 - creating with Concert Technology (C++ API) **375**
 - development steps (C API) **329**
 - development steps (C++ API) **375**
 - error handling (C API) **338**
 - error handling (C++ API) **252**
- arc **735**
- architecture
 - C++ API **372**
 - Callable Library (C API) **505**
 - Java API **425**
- arguments
 - null pointers (C API) **530**
 - optional (C API) **530**
- array
 - constructing (Java API) **467**

- creating multi-dimensional (C++ API) **416**
- creating variables in (Java API) **435**
- extensible (C++ API) **386**
- using for I/O (C++ API) **417**

B

- backtracking
 - criteria for **1349**
 - node selection and **1421**
 - tolerance **1349**
- BarAlg **1334**
 - large objective values and **730**
 - log file and **710**
 - settings **726**
- BarColNz **1335**
- BarCrossAlg **1336**
- BarDisplay **1337**
- BarEpComp **1338**
- BarGrowth **1339**
- BarItLim **1340**
- BarMaxCor **1341**
- BarObjRng **1342**
- baropt Interactive Optimizer command **197**
- BarOrder **1343**
- BarQCPEpComp **1344**
- barrier
 - advanced start and **1329**
 - detecting unbounded optimal faces **1339**

- maximum absolute objective function **1342**
- barrier limit
 - absolute value of objective function **1342**
 - centering corrections **1341**
 - detecting unbounded optimal faces **1339**
 - growth **1339**
 - iterations **1340**
- barrier optimizer
 - algorithm **693**
 - algorithms and infeasibility **731**
 - availability in Interactive Optimizer **197**
 - barrier display parameter **726**
 - centering corrections **726**
 - column nonzeros parameter and density **719**
 - column nonzeros parameter and instability **728**
 - corrections limit **726**
 - growth parameter **729**
 - infeasibility analysis **731**
 - inhibiting dual formulation **661**
 - linear **731**
 - log file **703**
 - numeric difficulties and **728**
 - numerical emphasis and **725**
 - parallel **1281**

- performance tuning **713**
 - preprocessing **718**
 - primal-dual **655**
 - QCP and **781**
 - quadratic **751, 774**
 - quadratic constraints and **781**
 - row-ordering algorithms **720**
 - second-order cone program (SOCP) and **782**
 - simplex optimizer and **696**
 - solution quality **711**
 - solving LP problems **691**
 - starting-point heuristics **721**
 - threads and **1277**
 - unbounded optimal face and **729**
 - unbounded problems **730**
 - uses **693**
- BarStartAlg **1345**
- BarStartAlg parameter
 - barrier starting algorithm **721**
- BarThreads removed **90**
- BAS file format **1501, 1546**
 - reading from Interactive Optimizer **215**
 - writing from Interactive Optimizer **207**
- basic variable
 - feasibility tolerance and **1379**
- basis

accessing information (C++ API) **267, 400**
advanced start and **1329**
advanced, primal feasible **651**
advanced, role in network **654**
advanced, starting from (example) **688**
basis information (Java API) **292**
column generation and **1057**
condition number **676, 681**
crash ordering and **1357**
crash parameter and **668**
crossover algorithms **695**
current (C API) **511**
differences between LP and network optimizers **747**
file formats for saving **1501, 1504**
from basis file **747**
infeasibility and **680**
Markowitz threshold and **1375**
maximum row residuals and **683**
network feasibility tolerance and **1413**
no factoring in network optimizer **747**
objective in Phase I and **680**
optimal and feasibility tolerance **1379**
optimal, condition number and **676**
optimal, numeric stability and **675**
parallel threads and advanced basis **651**

preprocessing versus starting (MIP) **865**
previous optimal (C API) **516**
refactoring rate **668**
removing objects from (C++ API) **407**
role in converting LP to network flow **747**
role in converting network-flow to LP **747**
role in network optimizer **742**
role in network optimizer to solve QP **742**
root algorithm and **1458**
saving best so far **677**
sensitivity analysis and (Java API) **461**
simplex iterations and **1393**
simplex refactoring frequency and **1452**
singularities and **677**
singularity repairs and **1470**
starting from advanced **1238**
starting from previous (C++ API) **276**
unexplored nodes in tree (MIP) **892**
unstable optimal **681**
basis file
 reading in Interactive Optimizer **215**
 writing in Interactive Optimizer **207**
BBInterval **1346**
best bound interval **1346**
best node

- absolute mip gap and **1370**
- backtracking and **1349**
- relative MIP gap and **1371**
- target gap and **1349**
- bibliography **366**
 - column generation **1057**
- Big M **1026**
- BndStrenInd **1347**
- Boolean parameter (C++ API) **274**
- Boolean variable
 - representing in model (C++ API) **247**
- bound
 - adding in Interactive Optimizer **218**
 - changing in Interactive Optimizer **227**
 - default values in Interactive Optimizer **176**
 - displaying in Interactive Optimizer **191**
 - entering in LP format in Interactive Optimizer **176**
 - removing in Interactive Optimizer **228**
 - sensitivity analysis in Interactive Optimizer **201**
- bound strengthening **1347**
- bound violation
 - feasibility (simplex) **1379**
 - FeasOpt **1380**
 - network flow **1413**
- bound violation (LP) **683**
- Bounded return status (Java API) **441**

- box variable in Interactive Optimizer **183**
- branch & cut algorithm
 - definition **821**
- branch direction **1348**
- branch variable selection callback **1263**
- branching
 - file format for entering direction **1510**
 - file format for entering priority **1510**
- branching direction (Java API) **455**
- branching, local **1394**
- BrDir **1348**
- breakpoint
 - discontinuous piecewise linear and **992**
 - example **989**
 - piecewise linear function and **989**
- BtTol **1349**
- BtTol parameter
 - backtrack parameter purpose **827**
- BZ2 file format **1501**

C

- call by value (C API) **522**
- Callable Library
 - categories of routines **506**
 - core **506**
 - debugging and **569**
 - description **132, 355**
 - example model **161**
 - parameters **541**

- using **503**
- Callable Library (C API) **315**
 - application development steps **329**
 - compiling and linking applications **319**
 - conceptual design **317**
 - CPLEX operation **323**
 - distribution file **320**
 - error handling **338**
 - opening CPLEX **325**
- callback
 - branch variable selection **1263**
 - control (definition) **1202**
 - control in parallel **1206**
 - cut **1261**
 - diagnostic (definition) **1194**
 - graphic user interface and **1221**
 - heuristic **1259**
 - incumbent **1265**
 - incumbent as filter in solution pool **967**
 - node selection **1266**
 - opportunistic parallel mode and **1188**
 - resetting to null (C API) **541**
 - resetting to null (C++ API) **395**
 - solve **1267**
 - using status variables **1221**
- callback reduced LP parameter **1396**
- callback, control **1405**

- callback, informational **92**
- candidate list limit (MIP) **1478**
- centering correction **1341**
- change Interactive Optimizer command **223**
 - bounds **227**
 - change options **224**
 - coefficient **229**
 - delete **231**
 - delete options **231**
 - objective **230**
 - rhs **230**
 - sense **226**
 - syntax **233**
- changing
 - bounds in Interactive Optimizer **227**
 - bounds setLB (Java API) **470**
 - bounds setUB (Java API) **470**
 - coefficients in Interactive Optimizer **229**
 - constraint names in Interactive Optimizer **225**
 - limit on barrier corrections **726**
 - maximization to minimization **745**
 - model setLinearCoef (Java API) **470**
 - objective in Interactive Optimizer **230**
 - parameters (C++ API) **274**
 - parameters in Interactive Optimizer **216**
 - pricing algorithm **740**

- problem in Interactive Optimizer **221**
- problem type
 - network to LP **747**
 - qp **763**
 - zeroed_qp **763**
- quadratic coefficients **763**
- righthand side (rhs) in Interactive Optimizer **230**
- sense in Interactive Optimizer **226**
- type of variable **1067**
- variable names in Interactive Optimizer **225**
- variable type (C++ API) **408**
- channel example **605**
- character string length requirements (C API) **532**
- check.c CPLEX file **533**
- Cholesky factor **714**
 - barrier iteration **693**
 - barrier log file and **708**
- choosing
 - optimizer (C API) **336**
 - optimizer (C++ API) **261**
 - optimizer in Interactive Optimizer **197**
- class library (Java API) **282**
- classpath (Java API) **284**
 - command line option **282**
- clean up data **233**
- clique cut **1351**
- clique cuts

- counting **849**
- definition **838**
- Cliques **1351**
- clock type parameter (Microsoft users) **44**
- ClockType **1352**
- cloneK.log **1287**
- clones **1287**
 - log files **1287**
 - threads and **1287**
- closing
 - application (C API) **518**
 - application (network) **745**
 - environment (C API) **518**
 - environment (network) **745**
 - log files **598**
- coefficient
 - changing in Interactive Optimizer **229**
- CoeRedInd **1353**
- ColReadLim **1354**
- column
 - dense **728**
 - density **719**
 - expressions (C++ API) **257**
 - index number (C API) **531**
 - name (C API) **531**
 - nonzeros parameter and density **719**
 - nonzeros parameter and instability **728**
 - referencing (C API) **531**
- column generation

- basis and **1057**
- cutting plane method and **1057**
- reduced cost and (example) **1059**
- reduced cost to select next variable **1057**
- columnwise modeling (C API) **545**
- columnwise modeling (C++ API) **414**
- columnwise modeling (Java API)
 - IloMPSModeler and **432**
 - objective and **466**
 - ranges and **466**
- comma separated value (CSV) file format **1501**
- command
 - executing from operating system in Interactive Optimizer **234**
 - input formats in Interactive Optimizer **168**
 - Interactive Optimizer list **168**
- compiler
 - DNDEBUG option (C++ API) **252**
 - error messages (C++ API) **244**
 - Microsoft Visual C++ Command Line (C API) **322**
 - using with CPLEX (C++ API) **241**
- compiling
 - applications **150**
 - applications (C API) **319**
 - applications (C++ API) **241**
- complementarity **693**

- barrier optimizer and **693**
- convergence tolerance **729**
- unbounded models and **730**
- complementarity convergence
 - barrier (LP, QP) **1338**
 - barrier (QCP) **1344**
 - LP **1338**
 - QCP **1344**
 - QP **1338**
- Component Libraries
 - defined **132**
 - running examples **149**
 - verifying installation **149**
- Component Libraries (definition) **355**
- Concert Technology
 - accessing parameter values (C++ API) **395**
 - application development steps (C++ API) **375**
 - C++ classes **245**
 - C++ objects **239**
 - compiling and linking applications (C++ API) **241**
 - CPLEX design in (C++ API) **239**
 - creating application (C++ API) **375**
 - description **355**
 - design (C++ API) **372**
 - error handling (C++ API) **252, 409**
 - running examples (C++ API) **241**
 - solving problem with (C++ API) **372**

- using (C++ API) **369, 421**
- writing programs with (C++ API) **371**
- Concert Technology (C++ API) **237, 277**
- Concert Technology Library
 - description **132**
 - example model **157**
- concurrent optimizer **1282**
 - licensing issues **1274**
 - non-default parameters and **658**
 - parallel processing and **1285**
 - root relaxation and **1285**
 - threads and **1277**
- cone (SOCP) **782**
- conflict
 - comparing IIS **1090**
 - definition **1087**
 - groups in **1119**
- conflict refiner **1085**
 - C++ API example **1114**
 - Interactive Optimizer example **1092**
 - MIP starts and **65, 1112**
- ConflictDisplay **1355**
- constraint
 - adding (C++ API) **272**
 - adding in Interactive Optimizer **218**
 - adding with user-written callback **1261**
 - changing names in Interactive Optimizer **225**

- changing sense in Interactive Optimizer **226**
- convex quadratic **775**
- creating (C++ API) **256**
- creating ranged (Java API) **432**
- cuts as **837**
- default names in Interactive Optimizer **176**
- deleting in Interactive Optimizer **231**
- displaying in Interactive Optimizer **189**
- displaying names in Interactive Optimizer **186**
- displaying nonzero coefficients in Interactive Optimizer **183**
- displaying number in Interactive Optimizer **183**
- displaying type in Interactive Optimizer **183**
- entering in LP format in Interactive Optimizer **175**
- indicator **1026**
- lazy **1134, 1261**
- lazy (LP) **1513**
- lazy (MPS) **1539**
- logical **1013**
- modeling linear (C++ API) **388**
- name limitations in Interactive Optimizer **176**
- naming in Interactive Optimizer **176**

- quadratic **775**
- range (C++ API) **256**
- ranged (Java API) **436**
- removing from basis (C++ API) **407**
- representing with IloRange (C++ API) **384**
- violation **683**
- constraints
 - adding to a model (Java API) **289**
- constructing arrays of variables (Java API) **467**
- continuous piecewise linear **992**
- continuous relaxation (Java API) **447**
- continuous relaxation (MIP) **821**
- continuous relaxation subproblem **1153**
- continuous variable
 - representing (C++ API) **247**
- control callback **1405**
 - definition **1202**
 - types of **1202**
- conventions
 - character strings (C API) **532**
 - naming **589**
 - notation **361**
 - numbering **580, 587**
 - numbering rows, columns **577**
- convergence tolerance
 - barrier algorithm and **658**
 - definition **714**
 - effect of tightness **728**

- performance and **714**
- convert CPLEX utility **591**
- converting
 - error code to string **745**
 - file formats **591**
 - network-flow model to LP **747**
 - network-flow problem to LP **747, 749**
- convex
 - quadratic constraints and **779**
- convex quadratic constraint **775**
- cover cut **1356**
- cover cut, flow **1382**
- cover cuts **839**
 - counting **849**
 - defined **839**
- Covers **1356**
- CPLEX
 - compatible platforms **132**
 - Component Libraries **132, 355**
 - core (C API) **506**
 - description **129**
 - directory structure **145**
 - installing **145**
 - licensing **148**
 - licensing (C++ API) **373**
 - parameters (C++ API) **395**
 - problem types **129**
 - quitting in Interactive Optimizer **235**
 - setting up **143**

- starting in Interactive Optimizer **167**
- technologies **132**
- cplex command in Interactive Optimizer **167**
- Cplex.Aborter **116**
- Cplex.CplexTime **67, 608**
- Cplex.EndTime **67, 608**
- cplex.h header file
 - C API **541**
 - extern statements in **540**
 - in an application **573**
 - macros for pointers in **536**
- cplex.jar (location) **282**
- cplex.log file
 - changing name **703**
 - clone logs **1287**
 - default name **596, 666**
- cplex.log file in Interactive Optimizer **197**
- Cplex.Ncliques removed **101**
- Cplex.Ncovers removed **101**
- Cplex.ReadMIPStart (deprecated) **32**
- Cplex.WriteMIPStart (deprecated) **32**
- CPX_CALLBACK_INFO_ENDTIME **67, 608**
- CPX_INTEGER_INFEASIBLE **1263**
- CPX_PARAM_ADVIND **1329**
 - MIP start **868**
 - presolve and advanced start **1246**
 - solution polishing and **857**
- CPX_PARAM_AGGCUTLIM **1331**

- CPX_PARAM_AGGFILL **1332**
- CPX_PARAM_AGGIND **1333**
- CPX_PARAM_BARALG **1334**
- CPX_PARAM_BARCOLNZ **1335**
- CPX_PARAM_BARCROSSALG **1336**
- CPX_PARAM_BARDISPLAY **1337**
- CPX_PARAM_BAREPCOMP **1338**
- CPX_PARAM_BARGROWTH **1339**
- CPX_PARAM_BARITLIM **1340**
- CPX_PARAM_BARMAXCOR **1341**
- CPX_PARAM_BAROBJRNG **1342**
- CPX_PARAM_BARORDER **1343**
- CPX_PARAM_BARQCPEPCOMP **1344**
- CPX_PARAM_BARSTARTALG **1345**
 - barrier starting algorithm **721**
- CPX_PARAM_BBINTERVAL **1346**
- CPX_PARAM_BNDSTRENIND **1347**
- CPX_PARAM_BRDIR **1348**
- CPX_PARAM_BTTOL **1349**
- CPX_PARAM_CLIQUES **1351**
- CPX_PARAM_CLOCKTYPE **1352**
 - example of parameter checking **541**
- CPX_PARAM_COEREDIND **1353**
- CPX_PARAM_COLREADLIM **1354**
- CPX_PARAM_CONFLICTDISPLAY **1355**
- CPX_PARAM_COVERS **1356**
- CPX_PARAM_CRAIND **1357**
- CPX_PARAM_CUTLO **1359**
 - conflict refiner and **1088**

FeasOpt and **1122**
CPX_PARAM_CUTPASS **1360**
CPX_PARAM_CUTSFACTOR **1361**
CPX_PARAM_CUTUP **1362**
 conflict refiner and **1088**
 FeasOpt and **1122**
CPX_PARAM_DATACHECK **1363**
 entering problem data and **533**
CPX_PARAM_DEPIND **1364**
 barrier **690**
CPX_PARAM_DISJ CUTS **1365**
CPX_PARAM_DIVETYPE **1366**
CPX_PARAM_DPRIIND **1367**
CPX_PARAM_EACHCUTLIM **1368**
CPX_PARAM_EPAGAP **1370**
CPX_PARAM_EPGAP **1371**
CPX_PARAM_EPINT **1372**
CPX_PARAM_EPMRK **1375**
CPX_PARAM_EPOPT **683, 1376**
CPX_PARAM_EPPER **1377**
CPX_PARAM_EPRELAX **1378**
CPX_PARAM_EPRHS **683, 1379**
CPX_PARAM_FEASOPTMODE **1380**
CPX_PARAM_FLOWCOVERS **1382**
CPX_PARAM_FLOWPATHS **1383**
CPX_PARAM_FPHEUR **1384**
CPX_PARAM_FRACCAND **1386**
CPX_PARAM_FRACCUTS **1387**
CPX_PARAM_FRACPASS **1388**

CPX_PARAM_GUBCOVERS **1389**
CPX_PARAM_HEURFREQ **1390**
CPX_PARAM_IMPLBD **1391**
CPX_PARAM_INTSOLLIM **1392**
CPX_PARAM_ITLIM **1393**
CPX_PARAM_LBHEUR **1394**
CPX_PARAM_LPMETHOD **1458**
 choosing LP optimizer **649**
 network flow **742**
CPX_PARAM_MEMORYEMPHASIS
1395
 barrier **716**
 conserving memory **672**
 final factor after preprocessing **662**
 presolve and **662**
CPX_PARAM_MIPCBREDLP **1396**
 branch callbacks and **1263**
 callback arguments and **1259**
 heuristic callbacks and **1259**
 incumbent callback and **1265**
 presolved and original problem **1258**
 user defined cuts and **1261**
CPX_PARAM_MIPDISPLAY **1398**
CPX_PARAM_MIPEMPHASIS **1400**
CPX_PARAM_MIPINTERVAL **1402**
CPX_PARAM_MIPORDIND **1403**
CPX_PARAM_MIPORDTYPE **1404**
CPX_PARAM_MIPSEARCH **1405**
CPX_PARAM_MIQCPSTRAT **1407**

CPX_PARAM_MIRCUTS **1409**
CPX_PARAM_MPSSLONGNUM **1410**
CPX_PARAM_NETDISPLAY **1411**
CPX_PARAM_NETEPOPT **1412**
CPX_PARAM_NETEPRHS **1413**
CPX_PARAM_NETFIND **1414**
CPX_PARAM_NETITLIM **1415**
CPX_PARAM_NETPPRIIND **1416**
CPX_PARAM_NODEFILEIND **1419**
 effect on storage **893**
 node files and **893**
CPX_PARAM_NODELIM **1420**
CPX_PARAM_NODESEL **1421**
CPX_PARAM_NUMERICALEMPHASIS
1422
 barrier **725**
 LP **674**
CPX_PARAM_NZREADLIM **1423**
CPX_PARAM_OBJDIF **1424**
CPX_PARAM_OBJLLIM **1425**
CPX_PARAM_OBJULIM **1426**
CPX_PARAM_PARALLELMODE **1427**
CPX_PARAM_PERIND **1430**
CPX_PARAM_PERLIM **1431**
CPX_PARAM_POLISHAFTEREPAGAP
1432
CPX_PARAM_POLISHAFTEREPGAP
1433

CPX_PARAM_POLISHAFTERINTSOL
1434
CPX_PARAM_POLISHAFTERNODE
1435
CPX_PARAM_POLISHAFTERTIME **1436**
CPX_PARAM_POLISHTIME
 solution polishing **857**
CPX_PARAM_POLISHTIME (deprecated)
1437
CPX_PARAM_POPULATELIM **1438**
CPX_PARAM_PPRIIND **1440**
CPX_PARAM_PREDUAL **1441**
CPX_PARAM_PREIND **1442**
CPX_PARAM_PRELINEAR **1443**
 advanced MIP control and **1261**
 advanced presolve **1243**
CPX_PARAM_PREPASS **1444**
CPX_PARAM_PRESLVND **1445**
CPX_PARAM_PRICELIM **1446**
CPX_PARAM_PROBE **1447**
 MIP **833**
CPX_PARAM_PROBETIME **1448**
CPX_PARAM_QPMAKEPSDIND **1449**
CPX_PARAM_QPMETHOD **1460**
 network flow and quadratic objective
 742
CPX_PARAM_QPNZREADLIM **1450**
CPX_PARAM_REDUCE **1451**
 advanced presolve **1242**

lazy constraints and **1137**
lazy constraints and advanced MIP control **1261**
optimal basis and **1246**
presolve and problem modifications **1249**
 problem modifications and **1246**
CPX_PARAM_REINV **1452**
CPX_PARAM_RELAXPREIND **1453**
 advanced presolve **1236**
CPX_PARAM_RELOBJDIF **1454**
CPX_PARAM_REPAIRTRIES **1455**
 MIP starts and **868**
CPX_PARAM_REPEATPRESOLVE **1456**
 purpose **865**
CPX_PARAM_RINSHEUR **1457**
CPX_PARAM_ROWREADLIM **1463**
CPX_PARAM_SCAIND **668, 1464**
CPX_PARAM_SCRIND **1465**
 error checking and **533**
 example lpex6.c **690**
 example with callbacks **1221**
 managing input and output **603**
 network flow **745**
 programming practices and **575**
 repeated singularities and **677**
CPX_PARAM_SCRIND parameter
 data checking and **533**
 reporting repeated singularities **677**

CPX_PARAM_SIFTALG **1466**
CPX_PARAM_SIFTDISPLAY **1467**
CPX_PARAM_SIFTITLIM **1468**
CPX_PARAM_SIMDISPLAY **1469**
CPX_PARAM_SINGLIM **1470**
CPX_PARAM_SOLNPOOLGAP **1471**
CPX_PARAM_SOLNPOOLCAPACITY **1472**
CPX_PARAM_SOLNPOOLGAP **1474**
CPX_PARAM_SOLNPOOLINTENSITY **1475**
CPX_PARAM_SOLNPOOLREPLACE **1477**
CPX_PARAM_STARTALG **1461**
 controlling algorithm in initial relaxation (MIP) **831**
 initial subproblem and **898**
 parallel processing and barrier **1285**
CPX_PARAM_STRONGCANDLIM **1478**
CPX_PARAM_STRONGITLIM **1479**
CPX_PARAM_SUBALG **1417**
 controlling algorithm at nodes **831**
 node relaxations and **899**
CPX_PARAM_SUBMIPNODELIM **1480**
 solution polishing and **857**
CPX_PARAM_SYMMETRY **1481**
CPX_PARAM_THREADS **1482**
CPX_PARAM_TILIM **1485**
 solution polishing and **857**

CPX_PARAM_TRELIM **1486**
 effect on storage **893**
 node files and **893**
CPX_PARAM_TUNINGDISPLAY **1487**
CPX_PARAM_TUNINGMEASURE **1488**
CPX_PARAM_TUNINGREPEAT **1489**
CPX_PARAM_TUNINGTILIM **1490**
CPX_PARAM_VARSEL **1491**
CPX_PARAM_WORKDIR **1493**
 barrier **716**
 node file subdirectory **893**
 node files and **893**
CPX_PARAM_WORKMEM **1494**
 barrier **716**
 node files and **893**
CPX_PARAM_WRITELEVEL **64, 1495**
CPX_PARAM_ZEROHALFCUTS **1497**
CPX_PREREDUCE_DUALONLY **1242**
CPX_PREREDUCE_NO_PRIMALORDUAL
1242
CPX_PREREDUCE_PRIMALANDDUAL
1242
CPX_PREREDUCE_PRIMALONLY **1242**
CPX_SEMICONTR **979**
CPX_SEMIINT **979**
CPXaddchannel routine
 data types in Callable Library and **524**
 message handling and **603**
CPXaddcols routine

 example in C API **341**
 maintainable code and **562**
 memory management and **526**
 modifying problems **1249**
 modular data in C API **335**
 populating problem (C API) **327**
CPXaddfpdest routine
 example lpex5.c **605**
 file pointers and **536**
 message channels and **603**
CPXaddfuncdest routine
 example **605**
 function pointers and (C API) **536**
 message channels and **603**
CPXaddindcontr **1026**
CPXaddrows routine
 example **547**
 LP example in C API **341**
 memory allocation and (C API) **526**
 modular data in C API **335**
 modularity and **562**
 network example in C API **345**
 populating model in C API **327**
CPXaddusercuts **1243**
CPXbaropt **1246**
CPXbasicresolve **1246**
CPXCENVptr **524**
CPXCHANNELptr data type **524**
CPXCHARptr data type **536**

CPXcheckaddcols routine **533**
CPXcheckaddrows routine **533**
CPXcheckchgcoeflist routine **533**
CPXcheckcopyctype routine **533**
CPXcheckcopylp routine **533**
CPXcheckcopylpwnames routine **533**
CPXcheckcopyqsep routine **533**
CPXcheckcopyquad routine **533**
CPXcheckcopysysos routine **533**
CPXcheckvals routine **533**
CPXchgbd **1249**
CPXchgcoeflist routine **562**
 example in C API **341**
 modular data in C API **335**
 populating model in C API **327**
CPXchg mipstart (deprecated) **32**
CPXchgprobtype routine **905**
CPXchgqpcoef routine **763**
 changing quadratic terms **763**
 example **763**
CPXchg rhs **1249**
CPXcloneprob routine
 advanced preprocessing and **1246**
 copying node LPs **1259**
CPXcloseCPLEX routine
 example lpex6.c **690**
 example mipex2.c **909**
 example qpex1.c **773**
 example qpex2.c **774**

LP example in C API **341**
managing input and output **605**
MPS example in C API **343**
network example in C API **345**
network flow problems **745**
purpose **518**
purpose in C API **325**
CPXCLPptr **524**
CPXCNETptr **524**
CPXcopybase **1246**
CPXcopybase routine **690**
CPXcopyctype routine **979**
CPXcopyctype routine
 checking types of variables **569**
 example mipex1.c **905**
 specifying types of variables **806**
CPXcopylp routine **514, 562**
 building model in memory for C API
 335
 efficient arrays in C API **335**
 example in C API **345**
 not for changing model in C API **328**
 populating model in C API **327**
CPXcopy mipstart (deprecated) **32**
CPXcopynettolp routine **747**
CPXcopyorder routine **975**
CPXcopyprotected **1245, 1263**
CPXcopyquad routine **773**
CPXcopysysos routine

example mipex3.c **975**
CPXcopystart **1246**
 advanced presolved solution and **1246**
 crushing primal or dual solutions **1246**
CPXcreateprob **1228**
CPXcreateprob routine **774**
 data types and **524**
 example lpex6.c **690**
 example mipex1.c **905**
 example mipex2.c **909**
 example qpex1.c **773**
 LP example in C API **343**
 network example in C API **345**
 problem object (C API) **513**
 purpose in C API **326**
 role in application **547**
 use in C API **326**
CPXcutcallbackadd **1243, 1261**
CPXdelchannel routine **603, 605**
CPXdelfpdest routine **536, 603, 605**
CPXdelfuncdest routine **603, 605**
CPXdelindconstr **1026**
CPXdisconnectchannel routine **603**
CPXdualopt **1246**
CPXENVptr data type **524**
CPXERR_NEGATIVE_SURPLUS symbolic
constant **550**
CPXERR_PRESLV_INF **1244**
CPXERR_PRESLV_UNBD **1244**

CPXERR_PRESOLVE_BAD_PARAM
1137
cpxerror message channel **603, 605**
CPXfclose routine **536**
CPXFILEPtr data type **536**
CPXflushchannel routine **603**
CPXfopen routine **536, 597**
CPXfputs routine **536**
CPXfreepresolve **1242**
CPXfreeprob **1242**
CPXfreeprob routine **517, 690, 773, 774,**
909
 file format example in C API **343**
 LP example in C API **341**
 network example in C API **345**
 purpose in C API **326**
CPXgetcallbackglobalb **1259, 1261**
CPXgetcallbackglobalub **1259**
CPXgetcallbackincumbent **1259**
CPXgetcallbackinfo routine **535, 1218,**
1220, 1221
CPXgetcallbacklp **1259, 1261**
CPXgetcallbacknodeintfeas **1263**
CPXgetcallbacknodelb **1259**
CPXgetcallbacknodelp **1259**
CPXgetcallbacknodeub **1259**
CPXgetcallbacknodex **1261**
CPXgetcallbackorder **1263**
CPXgetcallbackpseudocosts **1263**

CPXgetchannels routine **524, 603, 605**
CPXgetchgparam **121**
CPXgetcolindex routine **531**
CPXgetcols routine **550**
CPXgetconflict **52**
CPXgetctype routine **808**
CPXgetdblparam routine **529, 541**
CPXgetdblquality routine **676, 684, 711**
CPXgeterrorstring routine **745, 1221**
 closing LP example in C API **341**
 opening LP example in C API **341**
CPXgetintparam routine **529, 541**
CPXgetintquality routine **711**
CPXgetmipstart (deprecated) **32**
CPXgetnumcols routine **527**
CPXgetobjval routine **345, 905**
CPXgetparamtype **121**
CPXgetredlp **1246**
CPXgetrowindex routine **531**
CPXgetrownam routine **527**
CPXgetslack routine **905**
CPXgetsolnpoolmipstart (deprecated) **32**
CPXgetsolnpoolnummipstarts (deprecated)
32
CPXgetsos routine **808**
CPXgetstat routine **905, 1229**
CPXgetstrparam routine **529, 541**
CPXgettime **67, 608**
CPXgetx routine **522, 905**

CPXinfodblparam routine **529, 541**
CPXinfointparam routine **529, 541**
CPXinfostrparam routine **529, 541**
cpxlog message channel **603**
CPXlpopt **773, 774**
CPXlpopt routine **547, 1221**
 LP example in C API **341**
 network example in C API **345**
CPXLPptr data type **524**
CPXmemcpy routine **537**
CPXmipopt **1246**
CPXmipopt routine **905, 909**
CPXmsg routine **325, 512, 536, 603, 605**
CPXmsgstr routine **537**
CPXmstwrite (deprecated) **32**
CPXmstwritesolnpool (deprecated) **32**
CPXmstwritesolnpoolall (deprecated) **32**
CPXNETaddarcs routine **745**
CPXNETaddnodes routine **745**
CPXNETcheckcopynet routine **533**
CPXNETchgobjsen routine **745**
CPXNETcreateprob routine **524, 745**
CPXNETdelnodes routine **745**
CPXNETfreeprob routine **745**
CPXNETprimopt routine **745, 749**
CPXNETptr data type **524**
CPXNETsolution routine **745**
CPXnewcols default upper bound **102**
CPXnewcols routine **547, 562**

- LP example in C API **341**
- modular data in C API **335**
- populating model in C API **327**
- CPXnewrows routine **341, 562**
 - example in C API **341**
 - modular data in C API **335**
 - populating model in C API **327**
- CPXopenCPLEX routine
 - data types and **524**
 - example lpex1.c **605**
 - example lpex6.c **690**
 - example netex1.c **745**
 - example qpex1.c **773**
 - example qpex2.c **774**
 - file format example in C API **343**
 - initializing environment **512**
 - LP example in C API **341**
 - managing input and output **603**
 - network example in C API **345**
 - parameters and **541**
 - purpose in C API **325**
 - role in application **547**
- CPXordwrite routine **975**
- CPXpreaddrows **1249**
- CPXpresolve **1246**
- CPXprimopt **1246**
- CPXprimopt routine **539, 905**
- CPXPROB_FIXEDMILP symbolic constant **905**

- CPXPUBLIC symbolic constant **536**
- CPXPUBVARARGS symbolic constant **536**
- CPXqpopt routine **773, 774**
- CPXreadcopymipstart (deprecated) **32**
- CPXreadcopyprob routine **514**
 - example in C API **343**
 - formatted data files in C API **327**
- CPXreadcopysos removed **94**
- cpxresults message channel **603**
- CPXsetbranchcallbackfunc **1263**
- CPXsetcutcallbackfunc **1261**
- CPXsetdblparam routine **529, 541**
- CPXsetdefaults routine **541**
- CPXsetheuristiccallbackfunc **1259**
- CPXsetintparam routine **341**
 - arguments of **541**
 - example lpex6.c **690**
 - example netex1.c **745**
 - parameter types and **529**
 - redirecting output to screen **575**
 - selecting root algorithm **898**
- CPXsetlogfile routine **598, 703**
 - channels and **603**
 - collecting messages **533**
 - file pointers and **536**
 - managing log files **597**
- CPXsetlpcallbackfunc routine **536, 1220, 1221**
- CPXsetmipcallbackfunc routine **536, 1220**

CPXsetnodecallbackfunc **1266**
CPXsetsolvecallbackfunc **1267**
CPXsetstrparam routine **529, 541**
CPXsetterminate **116**
CPXsolution routine **547, 905, 1229**
 LP example in C API **341**
 network example in C API **345**
CPXsosread removed **94**
CPXsoswrite removed **94**
CPXstrcpy routine **537**
CPXstrlen routine **537**
CPXVOIDptr data type **536**
cpxwarning message channel **603**
CPXwriteprob routine **547, 568, 569, 677, 975**
 LP example in C API **341**
 network example in C API **345**
 testing in C API **339**
CraInd **1357**
creating
 algorithm object (C++ API) **250, 254**
 application with Concert Technology (C++ API) **375**
 array of variables (Java API) **435**
 arrays of variables (Java API) **435**
 automatic log file in Interactive Optimizer **197**
 binary problem representation (C API) **339**

Boolean variables (Java API) **435**
constraint (C++ API) **256**
CPLEX environment **745**
environment (C API) **345**
environment object (C++ API) **246, 254**
log file **597**
model (IloModel) (C++ API) **247**
model (Java API) **289**
model objects (C++ API) **254**
modeling variables (Java API) **432, 435**
network flow problem object **745**
new rows (Java API) **464**
objective function (C++ API) **256**
objective function (Java API) **432**
optimization model (C++ API) **247**
problem files in Interactive Optimizer **203**
problem object (C API) **326, 345, 513**
 ranged constraints (Java API) **432**
crossover
 verifying barrier solutions **724**
CSV file format **1501**
cut
 cliques (MIP) **1351**
 constraint aggregation limit and **1331**
 covers (MIP) **1356**

- definition **837**
- disjunctive (MIP) **1365**
- flow cover **1382**
- flow path (MIP) **1383**
- fractional pass limit **1388**
- Gomory fractional candidate limit **1386**
- Gomory fractional generation **1387**
- GUB (MIP) **1389**
- implied bound **1391**
- limit by type **1368**
- limiting number of **1361**
- MIP display and **1398**
- mixed integer rounding (MIR) **1409**
- node limit and **1420**
- pass limit **1360**
- reapplying presolve and **1456**
- user defined (LP) **1513**
- user defined (MPS) **1538**
- user-defined and preprocessing **1443**
- zero-half **1497**
- cut callback **1261**
- cut-counting routines **102**
- CutLo **1359**
- CutLo parameter
 - branch & cut **823**
 - conflict refiner and **1088**
 - FeasOpt and **1122**
- cutoff tolerance **1349**

- CutPass **1360**
- cuts **839, 1153**
 - adding **848**
 - clique **838**
 - counting **849**
 - cover **839**
 - disjunctive **840**
 - dual reductions and **1243**
 - flow cover **841**
 - flow path **842**
 - Gomory fractional **843**
 - GUB cover **844**
 - implied bound **845**
 - local or global **1155**
 - MIR **846**
 - re-optimizing **848**
 - recorded in MIP node log file **877**
 - what are **835**
 - zero-half **847**
- cuts, purging **43**
- CutsFactor **1361**
- CutsFactor parameter
 - controlling cuts **850**
- cutting plane method **1057**
- CutUp **1362**
- CutUp parameter
 - branch & cut **823**
 - conflict refiner and **1088**
 - FeasOpt and **1122**

D

data

- entering (C API) **514**
- entering in Interactive Optimizer **178**
- entry options **136**

data cleaning tool (C API) **121**

data cleaning tool (Interactive) **122**

data types

- special (C API) **524**

DataCheck **1363**

debugging

- Callable Library and **569**
- diagnostic routines and (C API) **533**
- heap **578**
- Interactive Optimizer and **569**
- return values and **576**

definition **803**

degeneracy

- dual **895**
- stalling and **678**

delete goal stacks **1172**

deleting

- constraints in Interactive Optimizer **231**
- model objects (C++ API) **407**
- problem options in Interactive Optimizer **231**
- variables (Java API) **470**
- variables in Interactive Optimizer **231**

deleting nodes **1172**

dense column **719**

dense matrix

- reformulating QP **760**

DepInd **1364**

DepInd parameter

- barrier **718**
- LPs and **662**

deprecated ports **18**

destroying

- CPLEX environment (C API) **518**
- nodes **745**
- problem object (C API) **517**

determinism **110**

deterministic

- definition **1427**

deterministic search

- control callbacks and **1206**
- incumbent callbacks and **967**
- mixed integer programming (MIP) and **1277**
- query callbacks in dynamic search **1199**
- query callbacks in parallel **1194**
- time limits and **1277**

devex pricing **666**

diagnosing

- infeasibility (barrier) **731**
- infeasibility (LP) **679**
- infeasibility (preprocessor) **1079**

- infeasibility (QP) **767**
- infeasibility as conflict **1085**
- performance problems (LP) **671**
- unboundedness **1084**
- diagnostic callback
 - definition **1194**
 - types of **1194**
- diagnostic routine
 - C API **533**
 - log file and (C API) **533**
 - message channels and (C API) **533**
 - redirecting output from (C API) **533**
- diet model (Java API) **438**
- diff method (Java API) **435**
- dimensions, checking **579**
- directory installation structure **145**
- discontinuous piecewise linear **992**
 - breakpoints and **992**
 - segments and **992**
- DisjCuts **1365**
- disjunctive cut **1365**
- disjunctive cuts **840**
- display Interactive Optimizer command **181, 225**
 - options **181**
 - problem **181**
 - bounds **191**
 - constraints **189, 190**
 - names **186, 187, 188**

- options **181**
- stats **183**
- syntax **182**
- sensitivity **201**
 - syntax **202**
- settings **217**
- solution **199**
 - syntax **200**
- specifying item ranges **185**
 - syntax **191**
- displaying
 - barrier information **701**
 - barrier solution quality **711**
 - basic rows and columns in Interactive Optimizer **199**
 - basis condition **676**
 - bound infeasibilities **682**
 - bounds in Interactive Optimizer **191**
 - column-nonzeros parameter **728**
 - constraint names in Interactive Optimizer **186**
 - constraints in Interactive Optimizer **189**
 - infeasibilities on screen **682**
 - messages **605**
 - MIP information periodically **877**
 - network objective values **739**
 - network solution information **745**
 - network solution on screen **745**

nonzero constraint coefficients in Interactive Optimizer **183**
number of constraints in Interactive Optimizer **183**
objective function in Interactive Optimizer **190**
optimal solution in Interactive Optimizer **195**
optimization progress **1221**
parameter settings in Interactive Optimizer **217**
post-solution information in Interactive Optimizer **199**
problem dimensions **579**
problem in Interactive Optimizer **179**
problem options in Interactive Optimizer **181**
problem part in Interactive Optimizer **183**
problem statistics **579**
problem statistics in Interactive Optimizer **183**
reduced-cost infeasibilities **682**
sensitivity analysis (C API) **347**
sensitivity analysis in Interactive Optimizer **201**
simplex solution quality **711**
slack values in Interactive Optimizer **199**
solution quality **681**

solution values in Interactive Optimizer **199**
solutions on screen **605**
type of constraint in Interactive Optimizer **183**
variable names in Interactive Optimizer **186**
variables **587**
variables in Interactive Optimizer **183**
diversity filter
 accessing **960**
 counting number of **960**
 filter file for **964**
diversity measure **1557**
DiveType **1366**
documentation for IDE **123**
DPE file format **1501**
DPriInd **1367**
DUA file format **1501**
dual feasibility **693**
dual reduction **1079**
dual residual **682**
dual simplex optimizer
 as default in Interactive Optimizer **195**
 availability in Interactive Optimizer **197**
 finding a solution (C API) **341**
 perturbing objective function **678**
 selecting **652**
 stalling **678**

- dual values
 - accessing (Java API) **292**
 - accessing in Interactive Optimizer **199**
- dual variable
 - solution data (C++ API) **399**
- duality gap **693**
- dynamic search **91**
 - incumbent callbacks and **967**
 - informational callbacks and **1188**
 - query callbacks and **1194**
- dynamic search algorithm
 - building blocks of **821**
 - definition **821**
 - MIP performance and **821**

E

- EachCutLim **1368**
- effort level
 - incumbent and **870**
 - MIP starts and **870**
 - solution pool and **870**
- EMB file format **1502**
- emphasis
 - memory (barrier) **716**
 - memory (LP) **672**
 - numerical (barrier) **725**
 - numerical (LP) **674**
- empty goal **1162, 1170**
- end method
 - IloEnv C++ class **380**

- enter Interactive Optimizer command **173, 806**
 - bounds **176**
 - file formats and **1505**
 - maximize **175**
 - minimize **175**
 - subject to **175, 218**
 - syntax **174**
- entering **806**
 - bounds in Interactive Optimizer **176**
 - constraint names in Interactive Optimizer **176**
 - constraints in Interactive Optimizer **175**
 - example problem in Interactive Optimizer **173**
 - item ranges in Interactive Optimizer **185**
 - keyboard data in Interactive Optimizer **178**
 - LPs for barrier optimizer **697**
 - mixed integer programs (MIPs) **806**
 - network arcs **745**
 - network data **745**
 - network data from file **749**
 - network nodes **745**
 - objective function in Interactive Optimizer **175, 176**
 - objective function names in Interactive Optimizer **176**

- problem in Interactive Optimizer **171, 175**
- problem name in Interactive Optimizer **173**
- variable bounds in Interactive Optimizer **176**
- variable names in Interactive Optimizer **175**
- enumeration
 - Algorithm (C++ API) **392**
 - BasisStatus (C++ API) **400**
 - BoolParam (C++ API) **395**
 - IntParam (C++ API) **395**
 - NumParam (C++ API) **395**
 - Quality (C++ API) **403**
 - Status (C++ API) **398**
 - String Param (C++ API) **395**
- environment
 - constructing (C++ API) **380**
 - initializing (C API) **512**
 - multithreaded (C API) **512**
 - releasing (C API) **518**
- environment object
 - creating (C++ API) **246, 254**
 - destroying (C++ API) **246**
 - memory management and (C++ API) **246**
- environment-problem mismatch **102**
- EpAGap **1370**
- EpAGap parameter

- objective near zero and **890**
- EpGap **1371**
- EpGap parameter
 - when to change **890**
- EpInt **90, 1372**
- EpLin **1373**
- EpMrk **1375**
- EpOpt **1376**
- EpOpt parameter **683**
- EpPer **1377**
- EpRelax **1378**
- EpRHS **1379**
- EpRHS parameter **683**
- eq method (Java API) **436**
- equality constraints
 - adding to a model (Java API) **289**
- error
 - invalid encrypted key (Java API) **284**
 - no license found (Java API) **284**
 - NoClassDefFoundError (Java API) **284**
 - UnsatisfiedLinkError (Java API) **284**
- error checking
 - diagnostic routines for (C API) **533**
 - MPS file format **590**
 - problem dimensions **579**
- error code **58**
- error codes added **96**
- error codes removed **95**
- error handling

- compiler (C++ API) **244**
- in Concert Technology (C++ API) **409**
- license manager (C++ API) **244**
- linker (C++ API) **244**
- programming errors (C++ API) **252**
- querying exceptions **575**
- runtime errors (C++ API) **252**
- testing installation **149**
- testing installation (C++ API) **244**
- Error return status (C++) **398**
- Error return status (Java API) **441**
- example
 - adding rows to a problem (C API) **345**
 - Column Generation **1055**
 - columnwise modeling (C API) **545**
 - columnwise modeling (C++ API) **414**
 - conflict refiner (Interactive Optimizer) **1093**
 - creating multi-dimensional arrays (C++ API) **416**
 - Cutting Stock **1055**
 - entering a problem in Interactive Optimizer **173**
 - entering and optimizing a problem (C API) **341**
 - entering and optimizing a problem in C# **306**
 - FORTTRAN **538**
 - ilolpex2.cpp (C++ API) **263**
 - ilolpex3.cpp (C++ API) **271**

- lpex1.c (C API) **341**
- lpex1.cs **306**
- lpex2.c (C API) **343**
- lpex3.c (C API) **345**
- message handler **605**
- MIP node log file **877**
- MIP optimization **901**
- MIP problem from file **907**
- MIP with SOS and priority orders **973**
- modifying an optimization problem (C++ API) **271**
- network optimization **738**
- optimizing QP **769**
- output channels **605**
- Piecewise Linear **987**
- project staffing **1093**
- reading a problem file (C API) **343**
- reading a problem from a file (C++ API) **263**
- reading QP from file **773, 774**
- resource allocation **1093**
- rowwise modeling (C API) **545**
- rowwise modeling (C++ API) **413**
- running (C++ API) **241**
- running Callable Library (C API) **321**
- running Component Libraries **149**
- running from standard distribution (C API) **321**

- solving a problem in Interactive Optimizer **195**
- using arrays for I/O (C++ API) **417**
- exception handling (C++ API) **252**
- executing a goal **1154**
- executing operating system commands in Interactive Optimizer **234**
- exportModel method
 - IloCplex class (C++ API) **259**
- expression
 - building (C++ API) **381**
 - column **257**
 - editable (Java API) **435**
 - in ranged constraints (Java API) **437**
 - linear (C++ API) **381**
 - logical (C++ API) **381**
 - piecewise linear (C++ API) **381**
 - square method (Java API) **435**
 - sum method (Java API) **435**
 - using modeling variables to construct (Java API) **432**
- external variables (C API) **521**
- extra rim vectors **589**

F

- FailGoal **1160**
- false return value of IloCplex.solve (Java API) **291**
- feasibility
 - analysis and barrier optimizer **731**

- check **1155**
- dual **668, 693**
- network flows and **735**
- primal **693**
- progress toward **678, 735**
- feasibility pump **112**
- feasibility tolerance
 - default **683**
 - largest bound violation and **683**
 - network optimizer and **740**
 - range of **683**
 - reducing **681**
- Feasible return status (C++) **398**
- Feasible return status (Java API) **441**
- feasible solution (Java API) **291**
- FeasOpt **1121**
 - definition **1122**
 - displaying infeasibilities **1125**
 - example **1125**
 - invoking **1123**
 - lower objective limit **1378**
 - mode **1380**
 - preferences **1124**
- feasOpt method
 - Java API **462**
- FeasOpt solution information **115**
- FeasOpt status code **113**
- FeasOptMode **1380**
- file format

converting **591**
described **585**
example QP problem data (C API)
773
lazy constraints in LP format **1142**
lazy constraints in MPS format **1145**
MST and MIP restart **459**
read options in Interactive Optimizer
211
RLP **1504**
solution pool filters **964**
write options in Interactive Optimizer
205

file formats

BAS **1501**
BZ2 **1501**
CSV **1501**
DPE **1501**
DUA **1501**
EMB **1502**
GZ **1502**
LP **1502**
MIN **1502**
MPS **1502**
MST **1503, 1548**
NET **1503**
ORD **1503, 1550**
PPE **1503**
PRE **1503**

PRM **1503**
REW **1503, 1551**
SAV **1504**
XML **1504**
file name
extension **259**
extension in Interactive Optimizer
206, 213
file reading routines in Callable Library **506**
file writing routines in Callable Library **506**
flow cover cut **1382**
aggregation limit **1331**
flow cover cuts
defined **841**
flow path cut **1383**
flow path cuts
defined **842**
FlowCovers **1382**
FlowPaths **1383**
FORTRAN **538, 580**
FPHeur **1384**
FracCand **1386**
FracCand parameter
controlling cuts **850**
FracCuts **1387**
FracPass **1388**
FracPass parameter
controlling cuts **850**
fractional cut
candidate limit **1386**

- generation **1387**
- pass limit **1388**
- fractional cuts
 - defined **843**
- free row **589**
- free variable
 - MPS files and **589**
 - reformulating QPs and **760**

G

- gap, relative (MIP) **66**
- ge method (Java API) **436**
- generalized upper bound (GUB) cover cuts **844**
- getBasisStatus method
 - IloCplex Java class **461**
- getBasisStatuses method
 - IloCplex C++ class **400**
- getBoundSA method
 - IloCplex C++ class **401**
- getBoundSA method (Java API) **461**
- getCplexStatus method
 - IloCplex C++ class **398, 401**
 - IloCplex class (C++ API) **250**
- getCplexStatus method (Java API) **291**
- getDefault method
 - IloCplex C++ class **395**
- getDual method
 - IloCplex C++ class **399**
- getDual method (Java API) **460**

- getDuals method
 - IloCplex C++ class **399**
 - IloCplex class (C++ API) **254**
- getDuals method (Java API) **460**
- getInfeasibilities method
 - C++ API **1125**
 - example **1125**
 - used with FeasOpt **1125**
- getMax method
 - IloCplex C++ class **395**
- getMax method (Java API) **453**
- getMin method
 - IloCplex C++ class **395**
- getMin method (Java API) **453**
- getNumVar method
 - IloCplex class (Java API) **463**
- getObjSA method
 - IloCplex C++ class **401**
- getObjSA method (Java API) **461**
- getObjValue method
 - IloCplex C++ class **399**
 - IloCplex class (C++ API) **251**
- getParam method
 - IloCplex C++ class **395**
- getParam method (Java API) **453**
- getQuality method
 - IloCplex class **676, 711**
- getRange method
 - IloCplex class (Java API) **463**
- getRangeSA method (Java API) **461**

- getReducedCost method
 - IloCplex C++ class **399**
- getReducedCost method (Java API) **460**
- getReducedCosts method
 - IloCplex C++ class **399**
 - IloCplex class (C++ API) **254**
- getReducedCosts method (Java API) **460**
- getRHSSA method
 - IloCplex C++ class **401**
- getSlack method
 - IloCplex C++ class **399**
- getSlacks method
 - IloCplex C++ class **399**
 - IloCplex class (C++ API) **254**
- getStatus method
 - IloCplex C++ class **398**
 - IloCplex class (C++ API) **250, 254**
- getStatus method (Java API) **291**
- getStatuses method
 - IloCplex class **689**
- getValue method
 - IloCplex C++ class **399**
 - IloCplex class (C++ API) **251**
- getValues method
 - IloCplex C++ class **399**
 - IloCplex class (C++ API) **254**
- global variables (C API) **521**
- goal
 - And-goal **1159**
 - empty **1162**

- executing **1154**
- Fail goal **1160**
- global cuts and **1174**
- Or-goal **1158**
 - solution injection by **1176**
- goal stack **1170**
- Gomory fractional cut
 - candidate limit **1386**
 - generation **1387**
 - pass limit **1388**
- Gomory fractional cuts
 - defined **843**
- graphic user interface (GUI) **1221**
- greater than equal to constraints
 - adding to a model (Java API) **289**
- group **1119**
 - definition **1119**
 - example in conflict **1119**
- GUB
 - constraint **844**
- GUB cut **1389**
- GUBCovers **1389**

H

- handle class
 - definition (C++ API) **246**
 - empty handle (C++ API) **247**
- handling
 - errors (C API) **338**
 - errors (C++ API) **252**

- exceptions (C++ API) **252**
- head **735**
- header file **573**
- heap, debugging **578**
- help Interactive Optimizer command **168**
- HeurFreq **1390**
- heuristic
 - callback **1259**
 - definition **854**
 - frequency **1390**
 - local branching **1394**
 - node **855**
 - relaxation induced neighborhood
 - search (RINS) **856, 1457**
 - RINSHeur parameter **856**
 - solutions **1176**
 - starting point **721**
 - SubMIPNodeLim and RINS **856**
- histogram
 - column counts **706**
 - detecting dense columns **719**
- histogram in Interactive Optimizer **192**
- I
- ill-conditioned
 - basis **683**
 - factors in **684**
 - maximum dual residual and **684**
 - problem **681**
- IloAdd template class (C++ API) **418**

- IloAddable class (Java API)
 - active model **438**
 - modeling objects and **432**
- IloAddNumVar class (C++ API) **257**
- IloArray template class (C++ API) **386**
- IloColumn class
 - and example (Java API) **468**
- IloColumn.and method (Java API) **297**
- IloColumnArray class (Java API) **467**
- IloConstraint class (C++ API) **388**
- IloConversion class (C++ API) **381, 388, 408**
- IloConversion class (Java API) **470**
- IloCplex class
 - getBasisStatus method (Java API) **461**
 - getBasisStatuses method (C++ API) **400**
 - getBoundSA method (C++ API) **401**
 - getCplexStatus method (C++ API) **398, 401**
 - getDefault method (C++ API) **395**
 - getDual method (C++ API) **399**
 - getDuals method (C++ API) **399**
 - getMax method (C++ API) **395**
 - getMin method (C++ API) **395**
 - getObjSA method (C++ API) **401**
 - getObjValue method (C++ API) **399**
 - getParam method (C++ API) **395**
 - getQuality method **676, 711**

getReducedCost method (C++ API) **399**
getReducedCosts method (C++ API) **399**
getRHSSA method (C++ API) **401**
getSlack method (C++ API) **399**
getSlacks method (C++ API) **399**
getStatus method (C++ API) **398**
getStatuses method **689**
getValue method (C++ API) **399**
getValues method (C++ API) **399**
IloMPSModeler and (Java API) **432**
isDualFeasible method (C++ API) **398**
isPrimalFeasible method (C++ API) **398**
modeling objects and (Java API) **432**
notifying about changes to (C++ API) **406**
objects in user application (C++ API) **372**
PrimalPricing (Java API) **453**
setDefault method (C++ API) **395**
setParam method (C++ API) **395**
solve method (C++ API) **398, 401, 403, 406, 420**
writeBasis method **677**
IloCplex class (C++ API) **239**
exportModel method **259**
getCplexStatus method **250**

getDuals method **254**
getObjValue method **251**
getReducedCosts method **254**
getSlacks method **254**
getStatus method **250, 254**
getValue method **251**
getValues method **254**
importModel method **259, 265**
setParam method **261**
setRootAlgorithm method **266**
solve method **250, 254, 267, 269**
solving with **250**
IloCplex class (Java API) **280**
add modeling object **289**
addLe method **296**
addMinimize method **296**
numVarArray method **296**
prod method **296**
scalProd method **296**
sum method **296**
IloCplex.Aborter **116**
IloCplex.getCplexTime **67, 608**
IloCplex.getEndTime **67, 608**
IloCplex.getNcliques removed **100**
IloCplex.getNcovers removed **100**
IloCplex.getNodeAlgorithm removed **100**
IloCplex.getRootAlgorithm removed **100**
IloCplex.readMIPStart (deprecated) **32**
IloCplex.setRootAlgorithm removed **100**

IloCplex.writeMIPStart (deprecated) **32**
IloCplex::Aborter **116**
IloCplex::getCplexTime **67, 608**
IloCplex::getEndTime **67, 608**
IloCplex::getNcliques removed **99**
IloCplex::getNcovers removed **99**
IloCplex::getNodeAlgorithm removed **99**
IloCplex::getRay **46**
IloCplex::getRootAlgorithm removed **99**
IloCplex::isBound deprecated **99**
IloCplex::isFixed **99**
IloCplex::readMIPStart (deprecated) **32**
IloCplex::setNodeAlgorithm removed **99**
IloCplex::setRootAlgorithm removed **99**
IloCplex::writeMIPStart (deprecated) **32**
IloCplexModeler interface
 modeling objects (Java API) **432**
IloCPModeler class (Java API) **432**
IloEnv class **380**
 end method (C++ API) **380**
IloEnv class (C++ API) **246**
 end method **246**
IloException class (C++ API) **252**
IloExpr C++ class **381**
IloExpr class (C++ API) **247**
IloExtractable class (C++ API) **247**
ILOG License Manager
 examples **617**
 invoking **616**

ILOG License Manager (ILM) **148, 612**
 CPLEX (C++ API) and **373**
 types of **613**
ILOG_LICENSE_FILE environment
variable **148**
IloLinearNumExpr class (Java API) **289**
IloLPMatrix class (Java API) **464**
IloMaximize C++ function **385**
IloMinimize C++ function **385, 418**
IloMinimize function (C++ API) **247**
IloModel C++ class **385**
IloModel class
 add method (C++ API) **386, 406**
 add method (Java API) **470**
 remove method (C++ API) **386, 406**
 remove method (Java API) **470**
IloModel class (C++ API)
 add method **247**
 extractable **247**
 role **239**
IloModel class (Java API)
 column method **297**
 numVar method **297**
IloModeler class
 basic modeling (Java API) **435**
 creating modeling objects (Java API)
 432
 creating variables (Java API) **435**
IloMPModeler class
 creating variables (Java API) **435**

IloMPModeler class (Java API) **432**
 delete method **470**
IloNumArray C++ class **386**
IloNumArray class (C++ API) **254**
IloNumColumn class (C++ API) **257**
IloNumExpr class
 objective and (Java API) **437**
 ranged constraints and (Java API) **436**
 variables and (Java API) **435**
IloNumExpr class (Java API) **289**
IloNumVar (Java API) **47**
IloNumVar C++ class **381, 388**
IloNumVar class
 modeling objects and (Java API) **432**
IloNumVar class (C++ API) **257**
 columns and **257**
IloNumVarArray C++ class **381**
IloNumVarclass
 extension of IloNumExpr (Java API)
 435
IloObjective C++ class **388**
IloObjective class
 addable objects (Java API) **438**
 as modeling object (C++ API) **388**
 declaring (C++ API) **383**
 modeling by column (Java API) **466**
 setExpr method in QP term **763**
IloObjective class (C++ API) **257**
 setLinearCoef method **258**

IloObjective::setCoef removed **99**
IloObjectiveSense class
 example (Java API) **437**
 maximizing (Java API) **437**
 minimizing (Java API) **437**
 objective function and (Java API) **437**
iloqpex1.cpp example
 example
 iloqpex1.cpp **771**
IloRange class
 adding constraints (C++ API) **384**
 linear constraints and (C++ API) **388**
 modeling by column (Java API) **466**
 modeling objects and (Java API) **438**
IloRange class (C++ API)
 casting operator for **257**
 example **247**
 setLinearCoef method **258**
IloRange class (Java API)
 setExpr method **299**
IloRange::setCoef removed **99**
IloSemiContVar class **388**
IloSolver as factory (Java API) **426**
IloSOS1 C++ class **388**
IloSOS2 C++ class **388**
ImplBd **1391**
implied bound cut **1391**
implied bound cuts
 defined **845**
importModel method

- IloCplex class (C++ API) **259, 265**
- include file **573**
- incumbent **38**
 - accessing in .NET API **49**
 - accessing in C API **51**
 - accessing in C++ API **46**
 - accessing in Java API **47**
 - backtracking and **1349**
 - cutoff tolerance and **1349**
 - diving and **1366**
 - local branching heuristic and **1394**
 - multiple MIP starts and **38**
 - node **826**
 - relaxation induced neighborhood search (RINS) and **1457**
 - solution **826**
 - solution pool absolute gap and **1471**
 - solution pool and **38, 917**
 - solution pool relative gap and **1474**
 - target gap and **1349**
- incumbent callback **1265**
 - solution pool and **967**
- index number (C API) **531**
- indicator constraint **1014, 1512**
 - definition **1026**
 - restrictions **1030**
- indicator variable **1029**
- infeasibility
 - barrier optimizer and **731**

- conflicts and **1085**
- diagnosing in network flows **749**
- displaying on screen **682**
- dual **726, 731**
- interpreting results **682**
- maximum bound **682, 683**
- maximum reduced-cost **682, 683**
- network flow **735**
- network optimizer and **749**
- norms **709**
- primal **710, 726, 731**
- ratio in barrier log file **710**
- reports **680**
- scaling and **681**
- unboundedness and (LP) **682**
- unscaled **681**
- infeasible (Java API) **291**
- Infeasible return status (C++ API) **398**
- Infeasible return status (Java API) **441**
- infeasible solution
 - accessing information (Java API) **462**
 - analyzing (C++ API) **401**
- InfeasibleOrUnbounded
 - return status (C++ API) **398**
 - return status (Java API) **441**
- informational callback **92**
 - dynamic search and **1188**
- initializing
 - CPLEX environment **745**

- problem object **745**
- problem object (C API) **513**
- input operator (C++ API) **386**
- installing CPLEX **143**
 - testing installation **149**
- instantiating
 - CPLEX environment **745**
 - problem object **745**
 - problem object (C API) **513**
- integer parameter (C++ API) **274**
- integer solution
 - diving and **1366**
- integer solution limit **1392**
- integer variable **1529**
 - in MPS file format **1529**
 - optimizer used (C API) **336**
 - representing in model (C++ API) **247**
- integrality constraints **1153**
- integrality tolerance **90**
 - MIP **891**
 - parameter **891**
- integrated development environment (IDE) and documentation **123**
- Interactive Optimizer **165, 235**
 - changing problem type (MIP) **809**
 - command formats **168**
 - commands **168**
 - debugging and **569**
 - description **132, 355**

- entering problems **1505**
- entering problems in **1505**
- example model **156**
- experimenting with optimizers **565**
- improving application performance **568**
- quitting **235**
- saving problems **1506**
- saving problems in **1506**
- starting **167**
- testing code in **561**
- tuning tool **636**
- tuning tool time limit **634**
- interruption **116**
- IntSolLim **1392**
- INumVar (.NET API) **49**
- invalid encrypted key (Java API) **284**
- isDualFeasible method
 - IloCplex C++ class **398**
- isolated point **994**
- isPrimalFeasible method
 - IloCplex C++ class **398**
- iteration
 - barrier centering corrections and **1341**
- iteration limit
 - barrier **1340**
 - network **1415**
 - perturbation and (simplex) **1431**
 - refactoring of basis (simplex) and **1452**

sifting **1468**
simplex **1393**
strong branching and (MIP) **1479**
iteration log in Interactive Optimizer **195,**
197
ItLim **1393**

J

Java Native Interface (JNI) **280**
Java serialization **432**
Java Virtual Machine (JVM) **282**
javamake for Windows **283**

K

knapsack constraint
cover cuts and **839**
GUB cover cuts and **844**
knapsack problem with reduced cost in
objective **1059**

L

lazy constraint **1261**
definition **1134**
Interactive Optimizer and **1141**
LP **1513**
LP file format and **1142**
MPS **1539**
MPS file format and **1145**
pool **1133, 1147**
LBHeur **1394**
le method
in expressions (Java API) **436**

libformat (Java API) **283**
license
CPLEX (C++ API) **373**
runtime **612**
licensing
CPLEX **148**
limiting
network iterations **740**
strong branching candidate list **889**
linear expression (C++ API) **381**
linear objective function (C++ API) **388**
linear optimization **130**
linear relaxation
as initial subproblem in MIP **898**
MIP and coefficients at nodes **865**
MIP and preprocessing **865**
MIP and progress reports **877**
linker
error messages (C++ API) **244**
using with CPLEX (C++ API) **241**
linking
applications **150**
applications (C++ API) **241**
Callable Library (C API) applications
319
local branching heuristic **1394**
log file
adding to in Interactive Optimizer **216**
barrier optimizer **703**
Cholesky factor in **708**

- clones and **1287**
- closing **598**
- contents **666, 711**
- cplex.log in Interactive Optimizer **197**
- creating **597**
- creating in Interactive Optimizer **197**
- default **597**
- description **595**
- diagnostic routines and (C API) **533**
- iteration **673**
- iteration log in Interactive Optimizer **195, 197**
- naming **597**
- network **739**
- parallel MIP optimizer and **1287**
- parameter **597**
- records infeasibilities **682**
- records singularities **677**
- relocating **597**
- renaming **597**
- logical constraint **1013, 1014**
 - example in early tardy scheduling **1043**
- logical expression (C++ API) **381**
- lower cutoff tolerance **42**
- LP
 - barrier optimizer **691**
 - choosing algorithm (C++ API) **392**
 - creating a model **154**

- network optimizer **733**
- node (C++ API) **261**
- problem format **130**
- problem formulation **356, 693**
- root (C++ API) **261**
- solving **646, 731**
- solving a model **153**
- solving pure (C++ API) **261**
- LP file
 - format in Interactive Optimizer **175**
 - reading in Interactive Optimizer **212**
 - writing in Interactive Optimizer **206**
- LP file format **1502, 1507, 1513**
 - indicator constraints in **1512**
 - lazy constraints **1142**
 - QPs and **761**
 - row, column order **587**
 - special considerations **587**
 - syntax rules **1507**
 - user cuts **1142**
- lpex1.c
 - example (C API) **341**
- LPex1.java example **294**
- LPMETHOD parameter in Interactive Optimizer **195**

M

- makefile (Java API) **283**
- managing
 - log file **595**

- memory (LP) **672**
- memory (MIP) **892**
- Markowitz tolerance **677, 678, 1375**
 - default **678**
 - increasing to stay feasible **678**
 - maximum value **677**
 - numeric stability and **677**
 - pivots and **677**
 - slow progress in Phase I and **678**
- maximal cliques
 - recorded in MIP node log file **877**
- maximization
 - concave QPs **753**
 - lower cutoff parameter **890**
- maximization in LP problem in Interactive Optimizer **175**
- maximize method
 - objective functions and (Java API) **437**
- maximum bound infeasibility **683**
- maximum infeasibility rule
 - variable selection and **1491**
- maximum reduced-cost infeasibility **683**
- maximum row residual **683**
- memory **892**
 - allocating when reading files **1505**
- memory emphasis
 - barrier **716**
 - continuous (LP) **672**
- memory leaks (C++ API) **380**

- memory management
 - by environment object (C++ API) **246**
 - MIPs and **892**
 - performance in LP **672**
 - refactoring frequency and **673**
- memory management (Java API) **426**
- MemoryEmphasis **1395**
- MemoryEmphasis parameter
 - barrier **716**
 - conserving memory and **672**
 - final factor after preprocessing **662**
 - presolve and **662**
- message channel
 - diagnostic routines and (C API) **533**
- message handler (example) **605**
- MIN file format **1502**
- minimal covers
 - recorded in MIP node log file **877**
- minimization
 - convex QPs **753**
 - upper cutoff parameter **890**
- minimization in LP problem in Interactive Optimizer **175**
- minimize method
 - objective functions and (Java API) **437**
- minimum infeasibility rule
 - variable selection and **1491**
- MIP **801, 909**
 - active model (Java API) **447**

- bound strengthening **1347**
- changing variable type **811**
- description **130**
- memory problems and **892**
- optimizer in Interactive Optimizer **197**
- problem formulation **803**
- progress reports **877**
- relaxation algorithm **895**
- solving (C++ API) **261**
- subproblem algorithm **895**
- subproblems **895**
- supplying first integer solution **868**
- terminating optimization **817**
- MIP callback reduced LP parameter **1396**
- MIP gap tolerance **817**
 - absolute **817**
 - relative **817**
- MIP limit
 - aggregation for cuts **1331**
 - cut by type **1368**
 - cuts **1361**
 - cutting plane passes **1360**
 - Gomory fractional cut candidates **1386**
 - nodes explored in subproblem **1480**
 - passes for Gomory fractional cuts **1388**
 - polishing time (deprecated) **1437**
 - probing time **1448**

- repair tries **1455**
- size of tree **1486**
- solutions **1392**
- termination criterion **1420**
- MIP solution information **115**
- MIP start
 - conflict refiner and **65, 1112**
 - continuous variables in **34**
 - effort level **870**
 - interaction with incumbent **868**
 - interaction with solution pool **868**
 - multiple **868**
 - multiple in MST file **63**
 - multiple, impact of **31**
 - query methods for **32**
 - solution pool and **38, 60, 948**
 - supplying first integer solution **868**
 - writing to file **1495**
 - writing to formatted files **64**
- MIP start values
 - file format for entering **1548**
- MIP strategy
 - backtracking **1349**
 - best bound interval **1346**
 - branch direction **1348**
 - branching variable **1491**
 - diving **1366**
 - heuristic frequency **1390**
 - local branching **1394**

- node algorithm **1417**
- node file management **1419**
- node selection **1421**
- presolve at nodes **1445**
- priority order **1403**
- probing **1447**
- quadratically constrained programs (MIQCP) **1407**
- RINS **1457**
- root algorithm **1461**
- strong branching and candidate limit **1478**
- strong branching and iteration limit **1479**
- MIP tree
 - advanced start and **1329**
- MIPDisplay **1398**
- MIPEmphasis **1400**
- MIPInterval **1402**
- mipopt Interactive Optimizer command **197**
- MIPOrdInd **1403**
- MIPOrdType **1404**
- MIPSearch **1405**
- MIPThreads removed **90**
- MIQCP **59**
- MIQCPStrat **1407**
- MIR cut **1409**
 - aggregation limit **1331**
- MIR cuts **846**

- MIRCuts **1409**
- Mixed Integer Linear Program (MILP)
 - definition **803**
 - definition (Java API) **447**
- Mixed Integer Programming (MIP)
 - definition **803**
- mixed integer programming (MIP)
 - determinism **1277**
 - parallel **1277**
 - solution pool and **913**
 - threads **1277, 1482**
- Mixed Integer Quadratic Program (MIQP)
 - definition **803**
 - definition (Java API) **447**
- Mixed Integer Quadratically Constrained Program (MIQCP) **803**
- mixed integer quadratically constrained programs (MIQCPs) **111**
- mixed integer rounding cut **1409**
- model
 - active (Java API) **438**
 - adding columns to **1066**
 - adding constraints (C++ API) **272**
 - adding objects (C++ API) **406**
 - adding submodels (C++ API) **385**
 - changing variable type **1067**
 - creating (C++ API) **247**
 - creating IloModel (C++ API) **247**
 - creating objects in (C++ API) **254**
 - deleting objects (C++ API) **407**

extracting (C++ API) **254, 390**
IloMPSModeler and (Java API) **432**
modifying (C++ API) **269**
modifying (Java API) **470**
notifying changes to IloCplex object
(C++ API) **406**
portfolio optimization **760**
reading from file (C++ API) **259, 265**
reformulating dense QP **760**
reformulating large QP **760**
removing objects (C++ API) **406**
serializing **592**
solving (C++ API) **267, 372, 391**
solving with IloCplex (C++ API) **420**
writing to file (C++ API) **259**
XML representation of **592**

modeling

- by columns (C++ API) **257**
- by columns (Java API) **297**
- by nonzeros (C++ API) **258**
- by nonzeros (Java API) **299**
- by rows (Java API) **296**
- columnwise (C API) **545**
- columnwise (C++ API) **414**
- objects (C++ API) **239, 372**
- rowwise (C API) **545**
- rowwise (C++ API) **413**
- variables (Java API) **289**

modeling by column (Java API)

IloMPSModeler and **432**
objective and **466**
ranges and **466**
modeling by rows (C++ API) **256**
modeling variable
 creating (Java API) **432**
 IloNumVar (Java API) **435**
modifying
 constraints in QCP **794**
 model (Java API) **470**
 problem object (C API) **328**
monitoring iteration log in Interactive
Optimizer **195**
MPS file format **1502, 1515, 1539**
 advanced basis in **1546**
 BAS file format **1501, 1546**
 BOUNDS section **1521**
 COLUMNS section **1519**
 conversion utility **591**
 CPLEX extensions **1527**
 data records **1518**
 DUA format **1501**
 example **1523**
 indicator records **1517**
 integer variables in **1529**
 lazy constraints in **1145**
 objective function name **1528**
 objective function sense **1528**
 proprietary information in **1503**

QUADOBJ section in **1533**
 quadratic coefficients in **1533**
 quadratically constrained program (QCP) and **1535**
 RANGES section **1520**
 REFROW section **1532**
 REW format **1503**
 RHS section **1519**
 ROWS section **1518**
 saving basis **1501**
 saving dual formulation **1501**
 saving embedded network **1502**
 saving modifications **590**
 saving QP **761**
 sense of rows **1518**
 SOS in **1531**
 user cuts in **1145**
 MPS file format in Interactive Optimizer **214**
 MPSLongNum **1410**
 MST file format **64, 1503, 1548**
 multiple MIP starts in **63**
 MST format
 advanced indicator parameter and **1548**
 multiple MIP starts
 conversion notes for **31**
 new feature **60**
 multithreaded application

 needs multiple environments (C API) **512**
N
 namespace conflicts (C API) **521**
 naming
 arcs in network flow **745**
 conventions **589**
 log file **597**
 node file **893**
 nodes in network flow **745**
 negative method
 expressions and (Java API) **435**
 negative semi-definite objective **753**
 NET file format **1503, 1540**
 NetDisplay **1411**
 NetEpOpt **1412**
 NetEpRHS **1413**
 NetFind **1414**
 NetItLim **740, 1415**
 netopt Interactive Optimizer command **197**
 NetPPriInd **1416**
 network
 converting to LP model **747**
 description **130**
 embedded **742**
 infeasibility in **735**
 modeling variables **735**
 problem formulation **735**
 network extractor **743**

network flow (C++ API) **272**
network object **734**
network optimizer **654, 733, 744**
 availability in Interactive Optimizer
 197
 file format for saving extracted
 network **1502**
 preprocessing and **744**
 problem formulation **735**
 solving with (C++ API) **272**
 turn off preprocessing **744**
new parameters **117**
Nmake (Java API) **283**
no license found (Java API) **284**
NoClassDefFoundError (Java API) **284**
node **1153**
 best estimate **1346**
 demand **735**
 from **735**
 head **735**
 presolve and **1445**
 sink **735, 738**
 source **735, 738**
 supply **735**
 tail **735**
 to **735**
 transshipment **735**
 viable **1259**
node file **893**

 compression of **1419**
 cpx name convention **893**
 parameters and **893**
 using with MIP **832**
 when to use **832, 893**
node heuristic **855**
node limit parameter **90, 93**
node log **877**
node LP
 solving (C++ API) **261**
node problem **1153**
node relaxation in MIQCP strategy **1407**
node selection
 backtracking and **1421**
 best bound interval and **1346**
node selection callback **1266**
node selection strategy
 best estimate **895**
 depth-first search **895**
NodeAlg **1417**
 controlling algorithm at subproblems
 (MIP) **831**
NodeAlg parameter
 node relaxations and **899**
NodeFileInd **1419**
NodeFileInd parameter
 effect on storage **893**
 node files and **893**
NodeLim **1420**
NodeSel **1421**

nonlinear expression
 definition **1021**
nonseparable **753**
nonzeros
 modeling (C++ API) **258**
 modeling (Java API) **299**
notation in this manual **140, 361**
notification (C++ API) **269**
notifying
 changes to IloCplex object (C++ API)
 406
null goal **1162**
 definition **1162**
 when to use **1162**
numbering conventions
 C **580**
 FORTRAN **580**
 row, column order **587**
numeric difficulties
 barrier growth parameter **729**
 barrier optimizer and **728**
 basis condition number and **676**
 complementarity **729**
 convergence tolerance **729**
 definition (LP) **674**
 dense columns removed **728**
 infeasibility and **678**
 sensitivity **676**
 unbounded optimal face **729**
numeric parameter (C++ API) **274**

numeric variable (C++ API) **388**
numerical emphasis
 barrier optimizer and **725**
 continuous (LP) **674**
NumericalEmphasis **1422**
NumericalEmphasis parameter
 barrier **725**
 LP **674**
numVarArray method (Java API) **296**
NzReadLim **1423**

O

ObjDif **1424**
ObjDif tolerance parameter **824**
objective
 current and backtracking **1349**
objective coefficients
 crash parameter and **668**
 modified in log file **739**
 network flows and **739**
 priority and **873**
objective difference
 absolute **824, 890, 1424**
 relative **824, 890, 1454**
objective function
 accessing value in Interactive
 Optimizer **199**
 accessing value of (C++ API) **399**
 adding to model (C++ API) **247**

changing coefficient in Interactive Optimizer **230**
changing sense **745**
changing sense in Interactive Optimizer **226**
constructor (Java API) **437**
creating (C++ API) **256**
creating (Java API) **432**
default name in Interactive Optimizer **176**
displaying in Interactive Optimizer **190**
entering in Interactive Optimizer **176**
entering in LP format in Interactive Optimizer **175**
free row as **589**
in log file **739**
in MPS file format **590, 1528**
maximization **590**
maximize (C++ API) **385**
minimize (C++ API) **385**
modeling (Java API) **437**
name in Interactive Optimizer **176**
network flows and **735**
optimality tolerance and **683**
preprocessing and **1079**
primal reductions and **1079**
quadratic coefficient in **1512**
representing with IloObjective (C++ API) **383**

sensitivity analysis in Interactive Optimizer **201**
sign reversal in **590**
objective value
 accessing slack in (C++ API) **399**
 in log file **739**
 network flows and **735**
 object range parameter **730**
 unbounded **730**
ObjLLim **1425**
ObjULim **1426**
operator() (C++ API) **257**
operator+ (C++ API) **257**
opportunistic mode
 MIP in parallel and **1277**
 MIP parallel optimizer and **1284**
 parallel mode parameter **1277**
opportunistic
 definition **1427**
opportunistic search
 component libraries and **1280**
 control callbacks in parallel **1206**
 informational callbacks and **1188**
 Interactive Optimizer and **1279**
 mixed integer programming (MIP) and **1277**
 query callbacks in parallel **1199**
 synchronization and **1277**
 thread safety and **1257**
 threads parameter and **1277**

- wait time and **1276**
- Optimal return status (C++ API) **398**
- Optimal return status (Java API) **441**
- optimal solution (Java API) **291**
- optimality
 - basis condition number and **676**
 - cutoff parameters **890**
 - infeasibility ration **710**
 - normalized error and **711**
 - singularities and **677**
 - tolerance **681, 683**
 - relative **890**
- optimality tolerance
 - absolute **890**
 - changing relative or absolute **890**
 - gap **890**
 - maximum reduced-cost infeasibility and **683**
 - Network and **740**
 - reducing **681**
 - relative **890**
 - relative, default **890**
 - setting **683**
 - when to change **890**
- optimization
 - interrupting **1221**
 - stopping **817, 1221**
- optimization model
 - creating (C++ API) **247**

- defining extractable objects (C++ API) **247**
- extracting (C++ API) **247**
- optimization problem
 - defining with modeling objects (C++ API) **372**
 - interrupting in Interactive Optimizer **198**
 - reading from file (C++ API) **264**
 - representing (C++ API) **254**
 - representing with IloModel (C++ API) **385**
 - solving with IloCplex (C++ API) **250**
- optimization routines in Callable Library **506**
- optimize Interactive Optimizer command **195**
 - re-solving **197**
 - syntax **196**
- optimizer
 - barrier (linear) **691, 731**
 - barrier (quadratic) **751, 774**
 - choosing (Java API) **446, 449**
 - choosing by problem type (C API) **336**
 - choosing by switch in application (C++ API) **266**
 - choosing in Interactive Optimizer **197**
 - concurrent **1282**

- differences between Barrier, simplex **696**
- dual simplex **652**
- MIP **801**
- network **654, 733, 744**
- options **134**
- parallel **1269**
- primal simplex **653**
- primal-dual barrier **655**
- syntax for choosing (C++ API) **261**
- optimizing
 - cuts **848**
- ORD file format **1503, 1550**
- ordering variables in Interactive Optimizer **188**
- OrGoal **1170**
- output
 - channel parameter **601**
 - debugging and **575**
 - redirecting **601**
- output method (Java API) **294**
- output operator (C++ API) **386**
- OutputStream (Java API) **294**

P

- parallel
 - choosing optimizers for **135**
 - license **1274**
 - optimizers **1269**
 - threads **1274**

- Parallel Barrier Optimizer **1281**
- parallel MIP optimization **110**
- Parallel MIP Optimizer **1283**
 - memory considerations **1286**
 - output log file **1287**
- parallel optimization
 - control callbacks and **1206**
 - query callbacks and **1194, 1199**
- parallel processing
 - root relaxation **1285**
 - selected starting algorithm **1285**
- parallelism
 - optimization mode **1427**
 - threads and **1482**
- ParallelMode **1427**
- parameter **57**
 - accessing
 - current value (C API) **541**
 - current value (C++ API) **395**
 - current value (Java API) **453**
 - default value (C API) **541**
 - default value (C++ API) **395**
 - default value (Java API) **453**
 - maximum value (C API) **541**
 - maximum value (C++ API) **395**
 - maximum value (Java API) **453**
 - minimum value (C API) **541**
 - minimum value (C++ API) **395**
 - minimum value (Java API) **453**

algorithmic **714**
barrier corrections **726**
Boolean (C++ API) **274**
Callable Library and **541**
changing (C++ API) **274**
changing in Interactive Optimizer **216**
classes of (Java API) **452**
displaying settings in Interactive Optimizer **217**
file format for **1503**
gradient **666**
integer (C++ API) **274**
list of settable in Interactive Optimizer **216**
log file **597**
NetFind network extractor **743**
numeric (C++ API) **274**
object range **730**
optimality cutoff **890**
output channel **601**
preprocessing dependency **718**
resetting to defaults in Interactive Optimizer **217**
routines in Callable Library **506**
screen indicator **745**
setting
 all defaults (C API) **541**
 all defaults (C++ API) **395**
 all defaults (Java API) **453**

branching direction (Java API) **455**
C API **541**
C++ API **395**
example algorithm (Java API) **449**
example steepest edge pricing (Java API) **453**
example turn off presolve (Java API) **453**
Java API **453**
priority in MIP (Java API) **455**
RootAlg (Java API) **449**
string (C++ API) **274**
symbolic constants as (C API) **541**
tree memory **892**
types of
 C API **541**
 C++ API **395**
 Java API string **452**
 Java API StringParam **452**
parameter specification file in Interactive Optimizer **217**
parameters as sets **120**
parameters, new **117**
path cut **1383**
path names in Interactive Optimizer **208**
performance
 Barrier
 centering corrections and **726**

- characteristics **693**
- dense columns and **719**
- memory management and **716**
- numeric difficulties and **728**
- preprocessing and **718**
- row order and **720**
- tuning **713**
- convergence tolerance and **714**
- LP
 - advanced basis and **664**
 - automatic selection of optimizer **651**
 - increasing available memory **672**
 - network as model **654**
 - numeric difficulties and **653**
 - parameters for **666**
 - preprocessing and **661**
 - preprocessing and memory **673**
 - refactoring and **673**
 - troubleshooting **671**
 - tuning **659**
- MIP
 - default optimizer and **814**
 - feasibility emphasis **815**
 - node files and **893**
 - probing and **833**
 - RINS and **856**
 - subproblems and **895**
 - swap space and **832**
 - troubleshooting **883**
 - tuning **819**
 - negative impact of Reduce parameter **1079**
 - Network optimizer
 - general observations **734**
 - tuning **740**
 - QP
 - reformulating for **760**
 - tuning **765**
 - SOS
 - branching strategies and **970**
 - PerInd **1430**
 - periodic heuristic **1390**
 - PerLim **1431**
 - perturbation constant (LP) **678**
 - perturbation constant (simplex) **1377**
 - perturbed problem
 - file format **1501, 1503**
 - perturbing
 - objective function **678**
 - variable bounds **678**
 - piecewise linear **987**
 - continuous **992**
 - definition **989**
 - discontinuous **992**
 - example **989**
 - example in early tardy scheduling **1043**

- expression (C++ API) **381**
- IloMPSModeler and (Java API) **432**
- isolated point ignored **994**
- steps **992**
- pivot selection **1375**
- PolishAfterEpAGap **1432**
- PolishAfterEpGap **1433**
- PolishAfterIntSol **1434**
- PolishAfterNode **1435**
- PolishAfterTime **1436**
- polishing solution **21**
 - example: first feasible **27**
 - example: gap **29**
 - example: time spent **25**
 - parameters for **22**
- PolishTime (deprecated) **1437**
- PolishTime parameter
 - solution polishing **857**
- pool
 - lazy constraints **1513**
 - of cuts **1135**
 - of lazy constraints **1135**
 - of solutions **913**
 - of user cuts **1135**
 - user-defined cuts **1513**
- populateByColumn method (Java API) **294**
- populateByNonzero method (Java API) **299**
- populateByNonzero method(Java API) **294**
- populateByRow (Java API) **294**

- PopulateLim **1438**
- populating problem object (C API) **514**
- populating problem object (network optimizer) **745**
- port changes **18**
- portability (C API) **536**
- portfolio optimization model **760**
- positive semi-definite
 - objective **753**
 - quadratic constraint **781**
 - second-order cone program (SOCP) and **782**
- PPE file format **1503**
- PPriInd **1440**
- PRE file format **1503**
- PreDual **1441**
- preference
 - example **1118**
 - FeasOpt **1124**
- PreInd **1442**
- PreLinear **1443**
- PrePass **1444**
- preprocessing
 - advanced basis and (LP) **664**
 - barrier and **718**
 - barrier optimizer **718**
 - definition of **661**
 - dense columns removed **728**
 - dependency parameter **718**

- dual reductions in **1079**
- lazy constraints and **1137**
- MIPs **865**
- network optimizer and **744**
- primal reductions in **1079**
- saving reduced problem **1503**
- second-order cone program (SOCP) and **782**
- simplex and **661**
- starting-point heuristics and **721**
- turning off **663**
- PreslvNd **1445**
- presolve **1236**
 - advanced start and **1329**
 - barrier preprocessing **718**
 - dependency checking in **662**
 - final factorization after uncrush in **662**
 - gathering information about **1246**
 - interface **1246**
 - lazy constraints and **1137**
 - limited **1246**
 - nodes and **1445**
 - process for MIP **1236**
 - protecting variables during **1245**
 - restricting dual reductions **1242**
 - restricting primal reductions **1242**
 - simplex and **661**
 - simplex preprocessing **661**
 - turning off (Java API) **453**

- presolved model
 - adding constraints to **1238**
 - building **1236**
 - freeing **1242**
 - freeing memory **1246**
 - retaining **1242**
- PriceLim **1446**
- pricing
 - candidate list limit **1446**
 - network **1416**
 - types available for dual simplex **1367**
 - types available in primal simplex **1440**
- pricing algorithms **740**
- primal feasibility **693**
- primal reduction **1079**
- primal simplex optimizer **653**
 - availability in Interactive Optimizer **197**
 - perturbing variable bounds **678**
 - stalling **678**
- primal variables **668**
- primal-degenerate problem **652**
- primopt Interactive Optimizer command **197**
- priority **873**
 - binary variables and **873**
 - integer variables and **873**
 - order **873**
 - parameter to control **873**
 - reading from file **873**

- semi-continuous variables and **873**
- semi-integer variables and **873**
- special ordered set (SOS) and **873**
- priority order
 - indicator **1403**
 - saving in ORD format **1503**
 - type to generate **1404**
- priority order (Java API) **455**
- PRM file format **1503**
- Probe **1447**
- Probe parameter
 - MIP **833**
- ProbeTime **1448**
- probing
 - MIP branching and **1447**
 - time limit **1448**
- probing parameter **833**
- problem
 - allocating memory when reading from file **1505**
 - analyzing infeasible (C++ API) **401**
 - change options in Interactive Optimizer **224**
 - changing in Interactive Optimizer **221**
 - creating binary representation (C API) **339**
 - data entry options **136**
 - displaying a part in Interactive Optimizer **183**

- displaying in Interactive Optimizer **179**
- displaying options in Interactive Optimizer **181**
- displaying statistics in Interactive Optimizer **183**
- entering from the keyboard in Interactive Optimizer **171**
- entering in Interactive Optimizer **1505**
- entering in LP format in Interactive Optimizer **175**
- format for saving presolved **1503**
- naming in Interactive Optimizer **173**
- reading files (C API) **343**
- saving in Interactive Optimizer **1506**
- solving (C API) **341**
- solving in Interactive Optimizer **193**
- solving with Concert Technology (C++ API) **372**
- verifying entry in Interactive Optimizer **181, 225**
- problem file
 - reading in Interactive Optimizer **209**
 - writing in Interactive Optimizer **203**
- problem formulation
 - barrier **693**
 - dual **693, 697**
 - ill-conditioned **681**
 - ilolpex1.cpp (C++ API) **254**
 - infeasibility reports **680**

- Interactive Optimizer and **173**
- linear **356**
- lpex1.c (C API) **341**
- lpex1.cs **306**
- LPex1.java (Java API) **294**
- network **735**
- network-flow **735**
- primal **693, 697**
- removing dense columns **719**
- standard notation for **130**
- switching from network to LP **747, 749**
- problem modification routines in Callable Library **506**
- problem object
 - creating (C API) **326, 513**
 - destroying (C API) **517**
 - freeing (C API) **517**
 - initializing (C API) **513**
 - instantiating (C API) **513**
 - modifying (C API) **328**
 - network **734**
 - populating **745**
 - populating (C API) **514**
- problem query routines in Callable Library **506**
- problem type
 - changing from network to LP **747, 749**

- changing to qp **763**
- changing to zeroed_qp **763**
- quadratic programming and **762**
- problem types solved by CPLEX **129**
- prod method in expressions (Java API) **435**
- pruned node **1153**
- PSD
 - positive semi-definite in objective function **753**
 - quadratic constraints and **781**
 - second-order cone program (SOCP) as exception to **782**
- pseudo cost
 - variable selection and **1491**
- pseudo reduced cost
 - variable selection and **1491**
- pseudo-shadow price
 - variable selection and **1491**

Q

- QCP **59**
 - barrier optimizer and **781**
 - convexity and **779**
 - description **130**
 - detecting problem type **784**
 - examples **797**
 - file types and **788**
 - modifying constraints in **794**
 - optimizer for **134**
 - PSD and **781**

QP

- applicable algorithms (C++ API) **261**
- description **130**
- example **769, 773, 774**
- portfolio optimization **760**
- problem formulation **753**
- reformulating large, dense models **760**
- solution example **773, 774**
- solving **751, 774**
- solving pure (C++ API) **261**
- QP relaxation **767**
- QPmakePSDInd **1449**
- QPNzReadLim **1450**
- QUADOBJ section in MPS files **1533**
- quadratic
 - constraints **775**
 - convex constraints **775**
- quadratic coefficient
 - changing **763**
- quadratic coefficients
 - in MPS file format **1533**
- quadratic objective function (C++ API) **388**
- quadratically constrained mixed integer program (MIQCP) **1407**
- quadratically constrained program (QCP)
 - MPS file format and **1535**
- quadratically constrained programming (QCP) **775**
- quit Interactive Optimizer command **235**
- quitting

ILOG CPLEX in Interactive Optimizer
235

Interactive Optimizer **235**

R

- range constraint
 - adding to a model (Java API) **289**
- range constraint (C++ API) **256**
- range filter
 - example **965**
- ranged constraint
 - creating (Java API) **432**
 - definition (Java API) **436**
 - name of (Java API) **436**
- ranged row **589**
- ray (C++ API) **46**
- re-solving in Interactive Optimizer **197**
- read Interactive Optimizer command **211, 212, 214**
 - basis files and **215**
 - file formats and **1505**
 - file type options **211**
 - syntax **215**
- reading
 - file format in Interactive Optimizer **211**
 - LP files in Interactive Optimizer **212**
 - MIP problem data **907**
 - MIP problem data from file **806**
 - model from file (C++ API) **259, 265**

MPS files in Interactive Optimizer **214**
network data from file **749**
problem files (C API) **343**
problem files in Interactive Optimizer
209
QP problem data from file **773, 774**
start values from MST file **868**

redirecting
diagnostic routines (C API) **533**
log file output **601**
output **575**
screen output **601**

Reduce **1451**

Reduce parameter
lazy constraints and **1137**

reduced cost
accessing (C++ API) **399**
accessing (Java API) **292, 460**
accessing in Interactive Optimizer **199**
choosing variables in column
generation **1059**
column generation and **1057**
pricing (LP) **666**

reduction
dual **1079**

reduction, primal **1079**

refactoring frequency
dual simplex algorithm and **658**
primal simplex algorithm and **658**

reference counting **1172**

reference row values **972**

refineConflict
Java API **462**

reflection scaling **743**

ReInv **1452**

relative gap
solution pool **1474**

relative gap (MIP) **66**

relative objective difference **824, 890, 1454**

relative optimality tolerance
default (MIP) **890**
definition **890**

relaxation
algorithm applied to **895**
of MIP problem **821**
QP **767**
solving MIPs (Java API) **447**

relaxation induced neighborhood search
(RINS) **856, 1457**

RelaxPreInd **1453**

RelaxPreInd parameter
advanced presolve **1236**

RelObjDif **1454**

RelObjDif tolerance parameter **824**

relocating log file **597**

remove method
IloModel C++ class **386, 406**

removed ports **18**

removing bounds in Interactive Optimizer
228

- renaming
 - log file **597**
- RepairTries **1455**
- RepairTries parameter
 - MIP starts and **868**
- RepeatPresolve **1456**
- RepeatPresolve parameter
 - purpose **865**
- representing optimization problem (C++ API) **254**
- residual
 - dual **682**
 - maximum dual **684**
 - maximum row **683**
 - row **682**
- return status
 - Bounded (Java API) **441**
 - Error (C++) **398**
 - Error (Java API) **441**
 - Feasible (C++) **398**
 - Feasible (Java API) **441**
 - Infeasible (C++) **398**
 - Infeasible (Java API) **441**
 - InfeasibleOrUnbounded (C++ API) **398**
 - InfeasibleOrUnbounded (Java API) **441**
 - Optimal (C++ API) **398**
 - Optimal (Java API) **441**
 - Unbounded (C++ API) **398**
 - Unbounded (Java API) **441**
 - Unknown (C++ API) **398**
 - Unknown (Java API) **441**
- return value
 - C API **527**
 - debugging with **576**
 - routines to access parameters (C API) **541**
- REW file format **1503, 1551**
- righthand side (RHS)
 - changing coefficient in Interactive Optimizer **230**
 - file formats for **589**
 - sensitivity analysis in Interactive Optimizer **201**
- rim vectors **589**
- RINSHeur **1457**
- RINSHeur parameter **856**
- RLP file format **1504**
- root LP
 - solving (C++ API) **261**
- root relaxation
 - parallel processing **1285**
- RootAlg **1458, 1460, 1461**
- RootAlg parameter
 - controlling initial relaxation algorithm **831**
 - initial subproblem and **898**
 - network flow **742**

- network flow and quadratic objective **742**
- parallel processing and barrier **1285**
- row
 - index number (C API) **531**
 - name (C API) **531**
 - referencing (C API) **531**
 - residual **682**
- row variable **1546**
- row-ordering algorithms **720**
 - approximate minimum degree (AMD) **720**
 - approximate minimum fill (AMF) **720**
 - automatic **720**
 - nested dissection (ND) **720**
- RowReadLim **1463**
- rowwise modeling
 - C API **545**
 - C++ API **413**

S

- SAV file format **761, 1504**
 - lazy constraints **1144**
 - user cuts **1144**
- SAV file format (C API) **345**
- saving
 - best factorable basis **677**
 - problem files in Interactive Optimizer **204**

- solution files in Interactive Optimizer **204**
- ScaInd **1464**
- ScaInd parameter **668**
- scaled problem statistics **682**
- scaling
 - alternative methods of **668**
 - definition **668**
 - feasibility and **681**
 - in network extraction **743**
 - infeasibility and **681**
 - maximum row residual and **683**
 - numeric difficulties and QP **767**
 - objective function in QP **767**
 - singularities and **677**
- scalProd method (Java API) **296**
- screen indicator **1465**
- screen indicator not available in this API **1465**
- search limit **1182**
- search tree **1153**
- second order cone programming **59**
- second order cone programming (SOCP) **775**
- second-order cone program (SOCP)
 - formulation **782**
- semi-continuous variable
 - C++ API **388**
 - example **982**
 - file format for entering **1511**

- Java API **432**
- priority and **873**
- semi-definite
 - negative and objective **753**
 - positive and constraints **781**
 - positive and objective **753**
- semi-integer variable **979**
- priority and **873**
- sense
 - changing in Interactive Optimizer **226**
- sensitivity analysis
 - performing (C API) **347**
 - performing in Interactive Optimizer **201**
- sensitivity analysis (C++ API) **401**
- sensitivity analysis (Java API) **461**
- separable **753**
- serialization **432**
- serializing **592**
- set Interactive Optimizer command **216**
 - advance **197**
 - available parameters **216**
 - defaults **217**
 - logfile **197**
 - simplex **195**
 - syntax **217**
- setDefault method
 - IloCplex C++ class **395**
- setExpr method
 - IloObjective class **763**

- setOut method **597**
- setOut method (Java API) **294**
- setParam method
 - IloCplex C++ class **395**
- setRootAlgorithm method
 - IloCplex class (C++ API) **266**
- setting
 - algorithm in LP (C++ API) **392**
 - all default parameters (C API) **541**
 - all default parameters (C++ API) **395**
 - callbacks to null (C API) **541**
 - callbacks to null (C++ API) **395**
 - parameters (C API) **541**
 - parameters (C++ API) **274**
 - parameters in C++ API **395**
 - parameters in Interactive Optimizer **216**
 - parameters to default in Interactive Optimizer **217**
- setWarning method (Java API) **294**
- shadow price
 - accessing in Intereactive Optimizer **199**
- SiftAlg **1466**
- SiftDisplay **1467**
- sifting **656**
 - iteration limit **1468**
 - node algorithm as **1417**
 - root algorithm as **1458**
- SiftItLim **1468**

SimDisplay **1469**

simplex

column generation and **1057**

dual **652**

feasibility tolerance in MIP **891**

iterations and candidate list **1479**

optimizer **696**

perturbation constant **1377**

pricing phase and **1057**

primal **653**

simplex limit

degenerate iterations **1431**

iterations **1393**

lower objective function **1425**

repairs of singularities **1470**

upper objective function **1426**

SingLim **1470**

singularity **677, 1470**

slack

accessing (Java API) **292**

accessing bound violations in (C++ API) **403**

accessing in constraints in active model (Java API) **443**

accessing in Interactive Optimizer **199**

accessing slack variables in constraints (C++ API) **399**

accessing slack variables in objective (C++ API) **399**

accessing values in Interactive Optimizer **199**

as indicator of ill-conditioning **684**

as reduced cost in infeasibility analysis **684**

example CPXgetslack **905**

in primal formulation (Barrier) **693**

in summary statistics **682**

infeasibilities as bound violations and **683**

infeasibility in dual associated with reduced costs **683**

maximum bound violation and (Java API) **463**

meaning in infeasible primal or dual LP **682**

pivoted in when constraint is removed (C++ API) **407**

reducing computation of steepest edge pricing **666**

role in inequality constraints (Barrier) **709**

role in infeasibility analysis **684**

using primal variables instead **668**

variable needed in basis (Network) **747**

variables and primal variables (dual) **668**

SOCP **59**

description **130**

- optimizer for **134**
- SOCP second-order cone program **782**
- SOL file format **64**
- SolnPoolAGap **1471**
- SolnPoolCapacity **1472**
- SolnPoolGap **1474**
- SolnPoolIntensity **1475**
- SolnPoolReplace **1477**
- solution
 - accessing basic rows and columns in Interactive Optimizer **199**
 - accessing quality information (C++ API) **403**
 - accessing values (C++ API) **251**
 - accessing values in Interactive Optimizer **199**
 - accessing values of (C++ API) **399**
 - alternative (MIP) **913**
 - basic infeasible primal **680**
 - basis **695**
 - complementary **693**
 - differences between barrier, simplex **696**
 - displaying basic rows and columns in Interactive Optimizer **199**
 - displaying in Interactive Optimizer **199**
 - example QP **773, 774**
 - feasible in MIPs **868**
 - incumbent **826**

- infeasible basis **731**
- midface **696**
- nonbasis **695**
- outputting (C++ API) **254**
- pool (MIP) **913**
- process in Interactive Optimizer **195**
- quality **690, 711, 726**
- querying results (C++ API) **251**
- reporting optimal in Interactive Optimizer **195**
- restarting in Interactive Optimizer **197**
- sensitivity analysis (C API) **347**
- sensitivity analysis in Interactive Optimizer **201**
- serializing **592**
- supplying first integer in MIPs **868**
- using advanced presolved **1246**
- verifying **724**
- writing to file **1495**
- XML representation of **592**
- solution file
 - creating **1551**
 - writing in Interactive Optimizer **203**
- solution polishing **21**
 - absolute gap as starting condition for **1432**
 - example: first feasible **27**
 - example: gap **29**
 - example: time spent **25**

integer solutions as starting condition for **1434**
nodes processed as starting condition for **1435**
parameters for **22**
relative gap as starting condition for **1433**
subproblems and **899**
time as starting condition for **1436**
solution pool **108, 1502**
absolute gap **1471**
capacity **1472**
changing fixed problem and **948**
counting members **40**
definition **913**
fixed problem and **948**
incumbent in **917**
indices in **37**
intensity **1475**
MIP start and **948**
multiple MIP starts and **38**
populate limit **1438**
replaced solutions **945**
replacement strategy **1477**
replacing existing solutions in **36**
relative gap **1474**
stopping criteria **926**
writing MIP start file **948**
solution pool status codes **114**

solve callback **1267**
solve method
IloCplex C++ class **398, 401, 403, 406, 420**
IloCplex class (C++ API) **250, 254, 267, 269**
solve method (Java API) **291, 294**
solving
diet problem (Java API) **443**
model (C++ API) **250, 267, 391**
node LP (C++ API) **261**
problem (C API) **341**
problem in Interactive Optimizer **193**
root LP (C++ API) **261**
single LP (Java API) **449**
subsequent LPs or QPs in a MIP (Java API) **450**
with network optimizer (C++ API) **272**
SOS
format for entering in MPS **1511**
in MPS file format **1531**
SOS file format removed **94**
sparse matrix
IloLPMatrix and (Java API) **464**
sparse matrix (C++ API) **272**
special ordered set (SOS)
role in model (Java API) **432**
type 1 (C++ API) **388**
type 2 (C++ API) **388**

- using **970**
- weights in **972**
- stalling **678**
- start, advanced **1329**
- starting
 - CPLEX in Interactive Optimizer **167**
 - from previous basis (C++ API) **276**
 - Interactive Optimizer **167**
 - new problem in Interactive Optimizer **173**
- starting algorithm
 - callbacks and **1232**
 - goals and **1232**
 - parallel processing **1285**
- static variables (C API) **521**
- status variables, using **1221**
- steepest-edge pricing **666, 893**
- step in piecewise linear function **992**
- stopping criterion
 - callbacks and **1221**
 - optimization (MIP) **817**
 - solution pools and **926**
- string parameter (C++ API) **274**
- strong branching **889**
 - candidate list and **1478**
 - iteration limit and **1479**
 - variable selection and **1491**
- strong thread limit removed **90**
- StrongCandLim **1478**

- StrongItLim **1479**
- structural variable **1546**
- structure of a CPLEX application (Java API) **288**
- SubMIPNodeLim **1480**
- SubMIPNodeLim parameter
 - RINS and **856**
 - solution polishing and **857**
- subproblem
 - definition (MIP) **831**
- summary statistics **682**
- Supported Platforms (Java API) **283**
- suppressing output to the screen **606**
- surplus argument (C API) **550**
- symbolic constants (C API) **528, 541**
- Symmetry **1481**
- symmetry breaking parameter **90**
- System.out method (Java API) **294**

T

- tail **735**
- target gap **1349**
- terminating
 - because of singularities **677**
 - MIP optimization **817**
 - network optimizer iterations **740**
- termination **116**
- termination criterion
 - barrier complementarity convergence (LP, QP) **1338**

barrier complementarity convergence (QCP) **1344**
barrier iterations **1340**
FeasOpt Phase I **1378**
MIP node limit **1420**
network iteration limit **1415**
simplex iteration limit **1393**
tree size (MIP) **1486**
tree size and memory **1419**
thread count parameter **90**
thread-safe (C API) **521**
Threads **1482**
threads **1274**
 clones **1287**
 parallel optimizers **1273**
 performance and **1275**
TiLim **1485**
TiLim parameter
 solution polishing and **857**
time
 as starting condition for solution polishing **1436**
time limit
 concurrent optimizer and **658**
 deterministic search and **1277**
 effects all algorithms invoked by concurrent optimizer **658**
 possible reason for Unknown return status (C++ API) **398**

 possible reason for Unknown return status (Java API) **441**
 TiLim parameter (MIP) **815**
time stamp **67, 608**
timing interface **67, 608**
tolerance
 absolute MIP gap **1370**
 absolute MIP objective difference **1424**
 absolute objective difference and **824**
 absolute optimality **890**
 backtracking (MIP) **1349**
 barrier complementarity convergence (LP, QP) **1338**
 basic variables and bound violation **1379**
 complementarity convergence QCP **1344**
 complementarity convergence, default of **729**
 complementary solution and **693**
 convergence and barrier algorithm **658**
 convergence and numeric inconsistencies **728**
 convergence and performance **714**
 cut callbacks and **1223**
 cut callbacks and (example) **1223**
 cutoff **1349**
 cutoff and backtracking **1349**
 cuts in goals and **1174**

default numeric (example LP) **675**
feasibility range **683**
feasibility (network primal) **1413**
feasibility (Network) **740**
feasibility and largest bound violation
683
feasibility default **683**
feasibility, reducing **681**
FeasOpt relaxation **1378**
integrality
 example (Java API) **470**
linearization **1373**
lower cutoff **1359**
Markowitz **677, 1375**
Markowitz and numeric difficulty **678**
Markowitz, increasing to stay feasible
678
MIP integrality **1372**
optimality **683**
optimality (Network) **740**
optimality (network) **1412**
optimality (simplex) **1376**
optimality, reducing **681**
relative MIP gap **1371**
relative MIP objective difference **1454**
relative objective difference and **824**
relative optimality **890**
relative optimality default **890**
role of (C++ API) **403**

role of (Java API) **463**
simplex optimality (example C++ API)
395
solution pool, absolute **1471**
solution pool, relative **1474**
termination and **817**
upper cutoff **1362**
violated constraints in goals and **1174**
warning about absolute and relative
objective difference **824**
when reducing does not help **681**
tranopt Interactive Optimizer command **197**
tree
 memory limit (MIP) **1486**
 MIP advanced start **1329**
TreLim **1486**
TreLim parameter
 effect on storage **893**
 node files and **893**
tuning
 measure **1488**
 repetition of **1489**
 reporting level **1487**
 time limit **1490**
tuning tool **109, 625**
TuningDisplay **1487**
TuningMeasure **1488**
TuningRepeat **1489**
TuningTiLim **1490**
type

changing for variable (Java API) **432**
conversion (Java API) **470**

U

unbounded (Java API) **291**
unbounded direction (C++ API) **46**
unbounded optimal face **1339**
 barrier optimizer **696**
 detecting **729**
Unbounded return status (C++ API) **398**
Unbounded return status (Java API) **441**
unboundedness
 dual infeasibility and **682**
 infeasibility and **682**
 infeasibility and (LP) **682**
 optimal objective and **682**
 unbounded ray and **1082**
UNIX
 building Callable Library (C API)
 applications **321**
 executing commands in Interactive
 Optimizer **234**
 installation directory **145**
 installing CPLEX **145**
 testing CPLEX (C++ API) **242**
 verifying installation **149**
Unknown return status (C++ API) **398**
Unknown return status (Java API) **441**
UnsatisfiedLinkError (Java API) **284**
unscaled problem statistics **682**

upper cutoff tolerance **42**
user cut
 definition **1134**
 Interactive Optimizer and **1141**
 MPS file format and **1145**
 pool **1133, 1147**
user defined cut (LP) **1513**
user defined cut (MPS) **1538**
utility routines in Callable Library **506**

V

variable
 accessing dual (C++ API) **399**
 Boolean (C++ API) **247**
 box in Interactive Optimizer **183**
 changing bounds in Interactive
 Optimizer **227**
 changing names in Interactive
 Optimizer **225**
 changing type (C++ API) **381, 408**
 changing type of **1067**
 constructing arrays of (Java API) **467**
 continuous (C++ API) **247**
 creating modeling (Java API) **432**
 deleting (Java API) **470**
 deleting in Interactive Optimizer **231**
 displaying in Interactive Optimizer
 183
 displaying names in Interactive
 Optimizer **186**

- entering bounds in Interactive Optimizer **176**
- entering names in Interactive Optimizer **175**
- external (C API) **521**
- global (C API) **521**
- in expressions (C++ API) **381**
- integer **1529**
- integer (C++ API) **247**
- modeling (Java API) **289, 435**
- name limitations in Interactive Optimizer **175**
- not addable (Java API) **438**
- numeric (C++ API) **388**
- order **587**
- ordering in Interactive Optimizer **188**
- removing bounds in Interactive Optimizer **228**
- removing from basis (C++ API) **407**
- representing with IloNumVar (C++ API) **381**
- row **1546**
- semi-continuous **1511**
- semi-continuous (C++ API) **388**
- semi-continuous (example) **982**
- semi-continuous (Java API) **432**
- semi-integer **979**
- static (C API) **521**
- structural **1546**

- type **806**
- variable selection
 - candidate list and **1478**
 - MIP strategy **1491**
 - simplex iterations and **1479**
- variable selection strategy
 - strong branching and best node progress **889**
 - strong branching and conservation of memory **895**
- variable type change (Java API) **432**
- variable, basic
 - feasibility tolerance and **1379**
- VarSel **1491**
- vectors, rim **589**
- violation
 - bound **683**
 - constraint **683**

W

- warning method (Java API) **294**
- wildcard
 - displaying ranges of items in Interactive Optimizer **185**
 - solution information in Interactive Optimizer **201**
- wildcard in Interactive Optimizer **185**
- Windows
 - building Callable Library (C API) applications **322**

- dynamic loading (C API) **322**
- installing CPLEX **145**
- Microsoft Visual C++ compiler (C API) **322**
- Microsoft Visual C++ IDE (C API) **322**
- testing CPLEX (C++ API) **243**
- verifying installation **149**
- WorkDir **1493**
- WorkDir parameter
 - barrier **716**
 - node file subdirectory **893**
 - node files and **893**
- working directory
 - barrier **716**
 - node files and **1419**
 - temporary files and **1493**
- working memory
 - barrier **716**
 - limit on **1494**
 - node files and **1419**
- WorkMem **892, 1494**
- WorkMem parameter
 - barrier **716**
 - node files and **893**
- write Interactive Optimizer command **204, 205, 206**
 - file formats and **1506**
 - file type options **205**
 - syntax **208**

- writeBasis method
 - IloCplex class **677**
- WriteLevel **64, 1495**
- writing
 - basis files in Interactive Optimizer **207**
 - file format for Interactive Optimizer **205**
 - LP files in Interactive Optimizer **206**
 - model to file (C++ API) **259**
 - problem files in Interactive Optimizer **203**
 - solution files in Interactive Optimizer **203**

X

- xecute Interactive Optimizer command **234**
 - syntax **234**
- XML
 - Concert Technology and **592**
 - serializing model, solution **592**
- XML file format **1504**
- XML schema change **41**

Z

- zero-half cut **847**
- zero-half cuts **1497**
- zero-ing out negligible coefficients (C API) **121**
- zero-ing out negligible coefficients (Interactive) **122**
- zero-ing out objective (Interactive) **122**

| ZeroHalfCuts **1497**